USENIX Association

Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)

July 7–9, 2025
Boston, MA, USA

© 2025 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-47-2

Symposium Organizers

Program Co-Chairs

Lidong Zhou, *Microsoft*
Yuanyuan Zhou, *University of California, San Diego*

Program Committee

Nitin Agrawal, *Google*
Adil Ahmad, *Arizona State University*
Sebastian Angel, *University of Pennsylvania*
Mahesh Balakrishnan, *Confluent, Inc.*
Daniel S. Berger, *Microsoft and University of Washington*
Shai Aviram Bergman, *Huawei Research Center Zurich*
Laurent Bindschaedler, *Max Planck Institute for Software Systems (MPI-SWS)*
James Bornholt, *The University of Texas at Austin and Amazon Web Services*
Thomas Bourgeat, *EPFL*
Marco Canini, *King Abdullah University of Science and Technology (KAUST)*
Ang Chen, *University of Michigan*
Haibo Chen, *Shanghai Jiao Tong University*
Kang Chen, *Tsinghua University*
Qi Chen, *Microsoft Research*
Rong Chen, *Shanghai Jiao Tong University*
Mosharaf Chowdhury, *University of Michigan*
Byung-Gon Chun, *FriendliAI and Seoul National University*
David Cock, *ETH Zurich*
Natacha Crooks, *University of California, Berkeley*
Heming Cui, *The University of Hong Kong*
Tiziano De Matteis, *Vrije Universiteit Amsterdam*
Daniele De Sensi, *Sapienza University of Rome*
Yufei Ding, *University of California, San Diego*
Mingkai Dong, *Shanghai Jiao Tong University*
Haggai Eran, *NVIDIA*
Ittay Eyal, *Technion—Israel Institute of Technology*
Pedro Fonseca, *Purdue University*
Ada Gavrilovska, *Georgia Institute of Technology*
Andreas Haeberlen, *University of Pennsylvania and Roblox*
Gernot Heiser, *University of New South Wales*
Michio Honda, *University of Edinburgh*
Jon Howell, *VMware Research*
Yu Hua, *Huazhong University of Science and Technology*
Ryan Huang, *University of Michigan*
Anand Iyer, *Georgia Institute of Technology*
Zhipeng Jia, *Google*
Sudarsun Kannan, *Rutgers University*
Rüdiger Kapitza, *Friedrich-Alexander-Universität Erlangen-Nürnberg*
Sanidhya Kashyap, *EPFL*
Vasileios Kemerlis, *Brown University*
Anne-Marie Kermarrec, *EPFL*
Samira Khan, *University of Virginia and Google*
Anurag Khandelwal, *Yale University*
Marios Kogias, *Imperial College London*
Dejan Kostic, *KTH Royal Institute of Technology*
Sanjeev Kumar, *Uber*
Fan Lai, *University of Illinois at Urbana–Champaign*
Andrea Lattuada, *Max Planck Institute for Software Systems (MPI-SWS)*
Patrick P. C. Lee, *The Chinese University of Hong Kong*
Philip Levis, *Stanford University and Google*
Amit Levy, *Princeton University*
Cheng Li, *University of Science and Technology of China*
Jialin Li, *National University of Singapore*
Shan Lu, *Microsoft Research*
Youyou Lu, *Tsinghua University*
Xiaosong Ma, *MBZUAI*
Martin Maas, *Google DeepMind*
Ratul Mahajan, *University of Washington*
Changwoo Min, *Igalia*
Kiran-Kumar Muniswamy-Reddy, *Amazon Web Services*
Srinivas Narayana, *Rutgers University*
Ravi Netravali, *Princeton University*
Amy Ousterhout, *University of California, San Diego*
Thomas Pasquier, *University of British Columbia*
Yuvraj Patel, *University of Edinburgh*
Gennady Pekhimenko, *University of Toronto and CentML*
Peter Pietzuch, *Imperial College London*
Clément Pit-Claudel, *EPFL*
Raluca Popa, *University of California, Berkeley*
Andi Quinn, *University of California, Santa Cruz*
Deepti Raghavan, *Brown University*
Ramachandran Ramjee, *Microsoft Research*
Luis Rodrigues, *INESC-ID and Instituto Superior Técnico, University of Lisbon*
Takahiro Shinagawa, *The University of Tokyo*
Anirudh Sivaraman, *New York University*
Muthian Sivathanu, *India*
Robert Soule, *Yale University*
Akshitha Sriraman, *Carnegie Mellon University and Google*
Patrick Stuedi, *Meta*
Michael Stumm, *University of Toronto*
Cheng Tan, *Northeastern University*
Shivaram Venkataraman, *University of Wisconsin—Madison*
Stephanie Wang, *University of Washington*
Hakim Weatherspoon, *Cornell University and Exosteller, Inc.*
Xingda Wei, *Shanghai Jiao Tong University*
Emmett Witchel, *The University of Texas at Austin*
Yubin Xia, *Shanghai Jiao Tong University*
Tianyin Xu, *University of Illinois at Urbana–Champaign*
Junfeng Yang, *Columbia University*
Eiko Yoneki, *University of Cambridge*
Haifeng Yu, *National University of Singapore*
Minlan Yu, *Harvard University*
Hong Zhang, *University of Waterloo*
Yang Zhou, *University of California, Davis, and University of California, Berkeley*

Poster Session Co-Chairs

Patrick P. C. Lee, *The Chinese University of Hong Kong*
Cheng Li, *University of Science and Technology of China*

Steering Committee

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

External Reviewers

Poulami Das
Peng Cheng
Jidong Zhai

Message from the OSDI '25 Program Co-Chairs

Welcome to the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25).

We're pleased to once again co-locate OSDI with the 2025 USENIX Annual Technical Conference (USENIX ATC '25). Together, these conferences present a broad and exciting collection of research papers, showcasing some of the most innovative work in the systems community. We hope you leave with new ideas, new collaborators, and new friends.

This year, OSDI received 339 submissions, of which 48 were accepted with shepherding. In addition, 5 papers from Revise-and-Resubmit track were later accepted, bringing the total to 53 papers included in the OSDI '25 proceedings — an overall acceptance rate of 16%. Three additional revise-and-resubmit papers are still under review and, if accepted, will appear in next year's OSDI '26 proceedings.

OSDI remains a single-track conference, allowing attendees to engage fully with every paper presented. We've organized a three-day program where all presentations are held in a single room, encouraging shared discussion and broad exposure to the entire program.

To fairly and thoroughly evaluate submissions across the wide range of topics OSDI covers, we assembled a program committee of 92 expert reviewers (not including the two of us as co-chairs). Reviewing such a large volume of high-quality research is no small task. Each paper received at least three reviews in the first round. A select group of papers advanced to a second round and received two or three additional reviews. We then held an intensive online discussion period followed by a two-day program committee meeting.

In total, the committee produced more than 1,500 reviews and thousands of discussion comments. We deeply appreciate the thoughtful and constructive feedback provided by the program committee. Reviewers also need to conduct additional reviews for revise-and-resubmit papers. Their careful assessments helped ensure that the selected papers reflect the best of what the systems community has to offer. Each accepted paper was assigned a shepherd from the committee to help the authors refine their work based on reviewer feedback, and we're proud of the clarity and quality of the final versions. We also extend our sincere thanks to the small group of external reviewers who generously contributed their expertise when additional perspectives were needed.

We also want to recognize the contributions of the Artifact Evaluation Committee, which was shared between OSDI and USENIX ATC. Co-chaired by Ryan Huang and Tianyin Xu—both of whom also served as OSDI '25 program committee members—this team assessed the supporting artifacts submitted with papers. Their work plays a key role in promoting reproducibility and encouraging rigorous experimental methods. One paper this year was recognized with a Distinguished Artifact Award.

The poster session at OSDI '25 was organized by Patrick P. C. Lee and Chen Li, who also served on the program committee. They coordinated the submission and selection process for this session, which includes posters from both accepted papers and additional research contributions. The poster session offers another opportunity for attendees to exchange ideas and connect with the broader systems community.

Once the accepted papers were decided, we began the process of deciding on the Jay Lepreau Best Paper Awards. A small committee of non-conflicted Program Committee members—excluding both program co-chairs— read all of the top-ranked and PC-nominated papers and made the final decision on the award papers. We sincerely thank the best paper selection committee for their thoughtful evaluations and dedication to recognizing outstanding contributions to the community—especially within a tight timeframe.

We are incredibly grateful to the many people who contributed to this conference. To the authors: thank you for submitting your work and for your dedication to pushing systems research forward. To the program committee and external reviewers: thank you for your time, expertise, and thoughtful evaluations. Special thanks to Eric Mugnier from UCSD and Carol Zhang from Microsoft Research for their behind-the-scenes help in the PC meeting, including conflict tracking and note-taking.

We also thank the USENIX staff for their exceptional support and professionalism. Their experience and coordination behind the scenes made the organizational process smoother at every step—from paper submissions to final conference.

Finally, thank you to all attendees. Conferences like OSDI thrive because of your participation, your questions, and your collaboration. We hope you enjoy the program, reconnect with colleagues, and discover new ideas and directions in systems research.

It has been an honor to serve as the Program Co-Chairs for OSDI '25. Thank you for trusting us with this role, and we hope the conference is rewarding and inspiring for you.

Warm regards,
Lidong Zhou
Yuanyuan (YY) Zhou
OSDI '25 Program Co-Chairs

19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)

July 7–9, 2025
Boston, MA, USA

Monday, July 7

Distributed Systems and Data Centers I

Basilisk: Using Provenance Invariants to Automate Proofs of Undecidable Protocols 1
Tony Nuda Zhang and Keshav Singh, *University of Michigan;* Tej Chajed, *University of Wisconsin-Madison;* Manos Kapritsos, *University of Michigan;* Bryan Parno, *Carnegie Mellon University*

Deriving Semantic Checkers from Tests to Detect Silent Failures in Production Distributed Systems 19
Chang Lou, *University of Virginia;* Dimas Shidqi Parikesit, *University of Virginia and Bandung Institute of Technology;* Yujin Huang, *The Pennsylvania State University;* Zhewen Yang and Senapati Diwangkara, *Johns Hopkins University;* Yuzhuo Jing, *University of Michigan;* Achmad Imam Kistijantoro, *Bandung Institute of Technology;* Ding Yuan, *University of Toronto;* Suman Nath, *Microsoft Research;* Peng Huang, *University of Michigan*

Picsou: Enabling Replicated State Machines to Communicate Efficiently 39
Reginald Frank, Micah Murray, Chawinphat Tankuranand, Junseo Yoo, Ethan Xu, and Natacha Crooks, *UC Berkeley;* Suyash Gupta, *University of Oregon;* Manos Kapritsos, *University of Michigan*

FineMem: Breaking the Allocation Overhead vs. Memory Waste Dilemma in Fine-Grained Disaggregated Memory Management 57
Xiaoyang Wang and Yongkun Li, *University of Science and Technology of China;* Kan Wu, *Google;* Wenzhe Zhu and Yuqi Li, *University of Science and Technology of China;* Yinlong Xu, *University of Science and Technology of China and Anhui Provincial Key Laboratory of High Performance Computing*

To PRI or Not To PRI, That's the question 75
Yun Wang, *Shanghai Jiao Tong University;* Liang Chen, Jie Ji, Xianting Tian, and Ben Luo, *Alibaba Group;* Zhixiang Wei, Zhibai Huang, and Kailiang Xu, *Shanghai Jiao Tong University;* Kaihuan Peng, Kaijie Guo, Ning Luo, Guangjian Wang, Shengdong Dai, Yibin Shen, and Jiesheng Wu, *Alibaba Group;* Zhengwei Qi, *Shanghai Jiao Tong University*

Enabling Efficient GPU Communication over Multiple NICs with FuseLink 91
Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, and Bowen Liu, *Hong Kong University of Science and Technology;* Han Tian, *University of Science and Technology of China;* Junxue Zhang, *Hong Kong University of Science and Technology;* Mingfei Wang, *MetaX Integrated Circuits;* Zhizhen Zhong, *Massachusetts Institute of Technology;* Guyue Liu, *Peking University;* Ying Zhang, *Meta;* Kai Chen, *Hong Kong University of Science and Technology*

Database Systems

Tigon: A Distributed Database for a CXL Pod 109
Yibo Huang, Haowei Chen, and Newton Ni, *The University of Texas at Austin;* Yan Sun, *University of Illinois Urbana–Champaign;* Vijay Chidambaram, Dixin Tang, and Emmett Witchel, *The University of Texas at Austin*

Mako: Speculative Distributed Transactions with Geo-Replication 129
Weihai Shen, *Stony Brook University;* Yang Cui, *Google;* Siddhartha Sen, *Microsoft Research;* Sebastian Angel, *University of Pennsylvania;* Shuai Mu, *Stony Brook University*

Quake: Adaptive Indexing for Vector Search 153
Jason Mohoney, Devesh Sarda, and Mengze Tang, *University of Wisconsin–Madison;* Shihabur Rahman Chowdhury and Anil Pacaci, *Apple;* Ihab F. Ilyas, *University of Waterloo;* Theodoros Rekatsinas, *Apple;* Shivaram Venkataraman, *University of Wisconsin–Madison*

Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD 171
Hao Guo and Youyou Lu, *Tsinghua University*

Skybridge: Bounded Staleness for Distributed Caches 187
Robert Lyerly, *Meta Platforms Inc.;* Scott Pruett, *unaffiliated;* Kevin Doherty and Greg Rogers, *Meta Platforms Inc.;* Nathan Bronson, *OpenAI;* John Hugg, *Meta Platforms Inc.*

AI + Systems I

KPerfIR: Towards a Open and Compiler-centric Ecosystem for GPU Kernel Performance Tooling on Modern AI Workloads 205
Yue Guan, *University of California, San Diego;* Yuanwei Fang, *Meta;* Keren Zhou, *George Mason University and OpenAI;* Corbin Robeck and Manman Ren, *Meta;* Zhongkai Yu, *University of California, San Diego;* Yufei Ding, *University of California, San Diego, and Meta;* Adnan Aziz, *Meta*

Mirage: A Multi-Level Superoptimizer for Tensor Programs 221
Mengdi Wu and Xinhao Cheng, *Carnegie Mellon University;* Shengyu Liu and Chunan Shi, *Peking University;* Jianan Ji and Man Kit Ao, *Carnegie Mellon University;* Praveen Velliengiri, *Pennsylvania State University;* Xupeng Miao, *Purdue University;* Oded Padon, *Weizmann Institute of Science;* Zhihao Jia, *Carnegie Mellon University*

QiMeng-Xpiler: Transcompiling Tensor Programs for Deep Learning Systems with a Neural-Symbolic Approach . . 239
Shouyang Dong, *University of Science and Technology of China, Cambricon Technologies, and Institute of Computing Technology, Chinese Academy of Sciences;* Yuanbo Wen, Jun Bi, Di Huang, and Jiaming Guo, *Institute of Computing Technology, Chinese Academy of Sciences;* Jianxing Xu and Ruibai Xu, *University of Science and Technology of China, Cambricon Technologies, and Institute of Computing Technology, Chinese Academy of Sciences;* Xinkai Song and Yifan Hao, *Institute of Computing Technology, Chinese Academy of Sciences;* Ling Li, *Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences;* Xuehai Zhou, *University of Science and Technology of China;* Tianshi Chen, *Cambricon Technologies;* Qi Guo, *Institute of Computing Technology, Chinese Academy of Sciences;* Yunji Chen, *Institute of Computing Technology, Chinese Academy of Sciences, and University of Chinese Academy of Sciences*

WaferLLM: Large Language Model Inference at Wafer Scale 257
Congjie He, Yeqi Huang, and Pei Mu, *University of Edinburgh;* Ziming Miao, Jilong Xue, Lingxiao Ma, and Fan Yang, *Microsoft Research;* Luo Mai, *University of Edinburgh*

Tuesday, July 8

AI + Systems II

BlitzScale: Fast and Live Large Model Autoscaling with *O*(1) Host Caching 275
Dingyan Zhang, Haotian Wang, Yang Liu, and Xingda Wei, *Shanghai Jiao Tong University;* Yizhou Shan, *Huawei Cloud;* Rong Chen and Haibo Chen, *Shanghai Jiao Tong University*

Bayesian Code Diffusion for Efficient Automatic Deep Learning Program Optimization. 295
Isu Jeong and Seulki Lee, *Ulsan National Institute of Science and Technology*

Training with Confidence: Catching Silent Errors in Deep Learning Training with Automated Proactive Checks. . . .313
Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang, *University of Michigan*

Neutrino: Fine-grained GPU Kernel Profiling via Programmable Probing 331
Songlin Huang and Chenshu Wu, *The University of Hong Kong*

Principles and Methodologies for Serial Performance Optimization 357
Sujin Park, Mingyu Guan, Xiang Cheng, and Taesoo Kim, *Georgia Institute of Technology*

Scheduling and Resource Management

Söze: One Network Telemetry Is All You Need for Per-flow Weighted Bandwidth Allocation at Scale 375
Weitao Wang and T. S. Eugene Ng, *Rice University*

Decouple and Decompose: Scaling Resource Allocation with DeDe 393
Zhiying Xu and Minlan Yu, *Harvard University;* Francis Y. Yan, *University of Illinois Urbana-Champaign*

Quantum Virtual Machines. 411
Runzhou Tao and Hongzheng Zhu, *University of Maryland, College Park;* Jason Nieh, *Columbia University;* Jianan Yao, *University of Toronto;* Ronghui Gu, *Columbia University*

QOS: Quantum Operating System 429
Emmanouil Giortamis, Francisco Romão, Nathaniel Tornow, and Pramod Bhatotia, *TU Munich*

Scalio: Scaling up DPU-based JBOF Key-value Store with NVMe-oF Target Offload 449
Xun Sun, Mingxing Zhang, Yingdi Shan, Kang Chen, and Jinlei Jiang, *Tsinghua University;* Yongwei Wu, *Tsinghua University, Quan Cheng Laboratory*

Distributed Systems and Data Centers II

Low End-to-End Latency atop a Speculative Shared Log with Fix-Ante Ordering 465
Shreesha G. Bhat, Tony Hong, Xuhao Luo, Jiyu Hu, Aishwarya Ganesan, and Ramnatthan Alagappan, *University of Illinois Urbana-Champaign*

Understanding Stragglers in Large Model Training Using What-if Analysis 483
Jinkun Lin, *New York University;* Ziheng Jiang, Zuquan Song, Sida Zhao, and Menghan Yu, *ByteDance Seed;* Zhanghan Wang, *New York University;* Chenyuan Wang, *ByteDance Seed;* Zuocheng Shi, *Zhejiang University;* Xiang Shi, *ByteDance;* Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, and Xin Liu, *ByteDance Seed;* Aurojit Panda and Jinyang Li, *New York University*

Fork in the Road: Reflections and Optimizations for Cold Start Latency in Production Serverless Systems 499
Xiaohu Chai, *Tsinghua University; Ant Group;* Tianyu Zhou, *Ant Group;* Keyang Hu, *Tsinghua University;* Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, Le Gao, Feng Yu, and Zhengyu He, *Ant Group;* Dong Du and Yubin Xia, *Shanghai Jiao Tong University;* Kang Chen, *Tsinghua University;* Yu Chen, *Quan Cheng Laboratory; Tsinghua University*

Kamino: Efficient VM Allocation at Scale with Latency-Driven Cache-Aware Scheduling. 519
David Domingo, *Rutgers University;* Hugo Barbalho and Marco Molinaro, *Microsoft Research;* Kuan Liu, Abhisek Pan, David Dion, and Thomas Moscibroda, *Microsoft Azure;* Sudarsun Kannan, *Rutgers University;* Ishai Menache, *Microsoft Research*

Zen: Empowering Distributed Training with Sparsity-driven Data Synchronization 537
Zhuang Wang, *Rice University;* Zhaozhuo Xu, *Stevens Institute of Technology;* Jingyi Xi, *unaffiliated;* Yuke Wang, Anshumali Shrivastava, and T. S. Eugene Ng, *Rice University*

Kernel and Operating Systems I

Extending Applications Safely and Efficiently. 557
Yusheng Zheng, *UC Santa Cruz;* Tong Yu, *eunomia-bpf Community;* Yiwei Yang, *UC Santa Cruz;* Yanpeng Hu, *ShanghaiTech University;* Xiaozheng Lai, *South China University of Technology;* Dan Williams, *Virginia Tech;* Andi Quinn, *UC Santa Cruz*

Tintin: A Unified Hardware Performance Profiling Infrastructure to Uncover and Manage Uncertainty 575
Ao Li, Marion Sudvarg, Zihan Li, Sanjoy Baruah, Chris Gill, and Ning Zhang, *Washington University in St. Louis*

Building Bridges: Safe Interactions with Foreign Languages through Omniglot 595
Leon Schuermann and Jack Toubes, *Princeton University;* Tyler Potyondy and Pat Pannuto, *University of California San Diego;* Mae Milano and Amit Levy, *Princeton University*

KRR: Efficient and Scalable Kernel Record Replay. 615
Tianren Zhang, *SmartX;* Sishuai Gong and Pedro Fonseca, *Purdue University*

Deterministic Client: Enforcing Determinism on Untrusted Machine Code 633
Zachary Yedidia, *Stanford University;* Geoffrey Ramseyer, *Stanford University and Stellar Development Foundation;* David Mazieres, *Stanford University*

Wednesday, July 9

Kernel and Operating Systems II

Disentangling the Dual Role of NIC Receive Rings. 651
Boris Pismenny, *EPFL & NVIDIA;* Adam Morrison, *Tel Aviv University;* Dan Tsafrir, *Technion – Israel Institute of Technology*

XSched: Preemptive Scheduling for Diverse XPUs. 671
Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

OS Rendering Service Made Parallel with Out-of-Order Execution and In-Order Commit 693
Yuanpei Wu and Dong Du, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education;* Chao Xu, *Fields Lab, Huawei Central Software Institute;* Yubin Xia and Yang Yu, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education;* Ming Fu, *Fields Lab, Huawei Central Software Institute;* Binyu Zang and Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education*

EMT: An OS Framework for New Memory Translation Architectures **711**
Siyuan Chai, Jiyuan Zhang, Jongyul Kim, Alan Wang, Fan Chung, and Jovan Stojkovic, *University of Illinois Urbana-Champaign;* Weiwei Jia, *University of Rhode Island;* Dimitrios Skarlatos, *Carnegie Mellon University;* Josep Torrellas and Tianyin Xu, *University of Illinois Urbana-Champaign*

Tiered Memory Management Beyond Hotness **731**
Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li, *Virginia Tech*

AI + Systems III

NanoFlow: Towards Optimal Large Language Model Serving Throughput **749**
Kan Zhu, *University of Washington;* Yufei Gao, *Tsinghua University and University of Washington;* Yilong Zhao, *University of Washington and University of California, Berkeley;* Liangyu Zhao, *University of Washington;* Gefei Zuo, *University of Michigan;* Yile Gu and Dedong Xie, *University of Washington;* Tian Tang and Qinyu Xu, *Tsinghua University and University of Washington;* Zihao Ye, Keisuke Kamahori, and Chien-Yu Lin, *University of Washington;* Ziren Wang, *Tsinghua University and University of Washington;* Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci, *University of Washington*

PipeThreader: Software-Defined Pipelining for Efficient DNN Execution **767**
Yu Cheng, Lei Wang, and Yining Shi, *School of Computer Science, Peking University;* Yuqing Xia, Lingxiao Ma, Jilong Xue, and Yang Wang, *Microsoft Research;* Zhiwen Mo, *Imperial College London and Microsoft Research;* Feiyang Chen, *Shanghai Jiao Tong University and Microsoft Research;* Fan Yang and Mao Yang, *Microsoft Research;* Zhi Yang, *School of Computer Science, Peking University*

WLB-LLM: Workload-Balanced 4D Parallelism for Large Language Model Training **785**
Zheng Wang, *University of California, San Diego, and Meta;* Anna Cai and Xinfeng Xie, *Meta;* Zaifeng Pan and Yue Guan, *University of California, San Diego;* Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, and Yuchen Hao, *Meta;* Yufei Ding, *University of California, San Diego, and Meta*

DecDEC: A Systems Approach to Advancing Low-Bit LLM Quantization **803**
Yeonhong Park, Jake Hyun, Hojoon Kim, and Jae W. Lee, *Seoul National University*

File and Storage Systems

Stripeless Data Placement for Erasure-Coded In-Memory Storage **821**
Jian Gao, Jiwu Shu, Bin Yan, and Yuhao Zhang, *Tsinghua University;* Keji Huang, *Huawei Technologies Co., Ltd*

PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection **839**
Hayley LeBlanc, *University of Texas at Austin;* Jacob R. Lorch and Chris Hawblitzel, *Microsoft Research;* Cheng Huang and Yiheng Tao, *Microsoft;* Nickolai Zeldovich, *MIT CSAIL and Microsoft Research;* Vijay Chidambaram, *University of Texas at Austin*

Fast and Synchronous Crash Consistency with Metadata Write-Once File System **859**
Yanqi Pan, Wen Xia, Yifeng Zhang, Xiangyu Zou, and Hao Huang, *Harbin institute of Technology, Shenzhen;* Zhenhua Li, *Tsinghua University;* Chentao Wu, *Shanghai Jiao Tong University*

Decentralized, Epoch-based F2FS Journaling with Fine-grained Crash Recovery **879**
Yaotian Cui and Zhiqi Wang, *The Chinese University of Hong Kong, China;* Renhai Chen, *College of Intelligence and Computing, Tianjin University, China;* Zili Shao, *The Chinese University of Hong Kong, China*

***Okapi*: Decoupling Data Striping and Redundancy Grouping in Cluster File Systems** **897**
Sanjith Athlur and Timothy Kim, *Carnegie Mellon University;* Saurabh Kadekodi, *Google;* Francisco Maturana and Xavier Ramos, *Carnegie Mellon University;* Arif Merchant, *Google;* K. V. Rashmi and Gregory R. Ganger, *Carnegie Mellon University*

Privacy and Security

Compass: Encrypted Semantic Search with High Accuracy **915**
Jinhao Zhu, *UC Berkeley;* Liana Patel, *Stanford University;* Matei Zaharia and Raluca Ada Popa, *UC Berkeley*

Weave: Efficient and Expressive Oblivious Analytics at Scale **939**
Mahdi Soleimani, Grace Jia, and Anurag Khandelwal, *Yale University*

Paralegal: Practical Static Analysis for Privacy Bugs. **957**
Justus Adam, Carolyn Zech, Livia Zhu, Sreshtaa Rajesh, Nathan Harbison, Mithi Jethwa, Will Crichton, Shriram Krishnamurthi, and Malte Schwarzkopf, *Brown University*

MettEagle: Costs and Benefits of Implementing Containers on Microkernels **979**
Till Miemietz, Viktor Reusch, and Matthias Hille, *Barkhausen Institut;* Lars Wrenger, *Leibniz-Universität Hannover;* Jana Eisoldt, *Barkhausen Institut;* Jan Klötzke, *Kernkonzept GmbH;* Max Kurze, *Technische Universität Dresden;* Adam Lackorzynski, *Technische Universität Dresden and Kernkonzept GmbH;* Michael Roitzsch, *Barkhausen Institut;* Hermann Härtig, *Barkhausen Institut and Technische Universität Dresden*



Basilisk: Using Provenance Invariants to Automate Proofs of Undecidable Protocols

Tony Nuda Zhang
University of Michigan

Keshav Singh
University of Michigan

Tej Chajed
University of Wisconsin–Madison

Manos Kapritsos
University of Michigan

Bryan Parno
Carnegie Mellon University

Abstract

Distributed protocols are challenging to design correctly. One promising approach to improve their reliability is to use formal verification to prove that a protocol satisfies a desired safety property. These proofs require finding an inductive invariant that holds in the initial states of the system, implies safety, and is inductive over state transitions. Devising an inductive invariant is a difficult task that prior work has either required the developer to find manually by a painful search process, or automated by constraining the protocol to a decidable but restrictive fragment of logic.

In this work, we aim to automatically find inductive invariants without restricting the logic. We achieve this with two key insights. First, many of the complex inter-host properties that prior work required the developer to provide can instead be expressed using *Provenance Invariants*, a class of invariants that relate a local variable in a host to its provenance, i.e., the protocol step that caused it to take on its current value. By tracing the provenance of one host variable back to another host's actions, we can derive an invariant relating the two hosts' states. Second, we develop an algorithm called atomic sharding to derive Provenance Invariants automatically by statically analyzing the protocol's steps.

We implement these ideas in a tool called Basilisk and apply it to 16 distributed protocols, including complex ones like Multi-Paxos. Basilisk automatically finds inductive invariants and proves their inductiveness, with little or no developer assistance. In all cases, these generated inductive invariants are sufficient for us to prove safety without needing to identify any new invariants.

1 Introduction

Distributed protocols are notoriously difficult to design and implement correctly. After decades of relying on testing to develop robust protocols, in the last ten years many researchers and developers have turned to formal verification as an attractive alternative [2, 16, 24, 30, 31, 33, 36]. In this approach, the developer typically proves that some desirable safety property (e.g., no two nodes hold a lock at the same time) is maintained throughout all executions of the protocol.

Unfortunately, a safety property is, on its own, often too weak to support an inductive argument. Instead, one typically proves safety by finding a stronger property, called an *inductive invariant*, that satisfies the following three conditions: (1) it implies the safety property, (2) it holds in the initial states of the system, and (3) it is maintained by all steps. The developer must find such an inductive invariant and prove it meets these conditions in order to complete their proof.

In recent years, researchers have proposed a series of algorithms and tools that derive and prove these invariants automatically, with little to no input from the developer [7, 9, 13–15, 21, 28, 37, 38]. However, such approaches share a major obstacle to their practicality: they require the target protocols to be written within a decidable logic, such as *effectively propositional reasoning* (EPR [29]). Expressing a protocol in EPR is itself a challenging endeavor [26]. As an example of its restrictiveness, EPR prohibits common programming patterns such as the use of arithmetic (e.g., $i := i + 1$).

Rejecting the restrictions of EPR allows the developer to write their protocols naturally, but such freedom comes at a cost. Working in an undecidable logic means that the developer must manually derive and prove an inductive invariant for each protocol. As many have observed [9, 21, 37, 38], this requires significant effort. A key difficulty stems from the iterative and creative nature of finding a correct inductive invariant. It is hard to know *a priori* what the clauses in an inductive invariant should be. Rather, the developer starts with their best guess, then repeatedly attempts to prove the inductiveness of their current guess, gradually narrowing in on a correct inductive invariant. Developing an invariant this way is labor intensive. For instance, the authors of IronFleet [10, 11] report spending months to identify and prove an inductive invariant for the Multi-Paxos protocol [18, 19].

Recently, Kondo [40] showed how to automatically find small portions of the inductive invariant. Unfortunately, much

of the remaining portions, particularly the conceptually difficult ones—those expressing inter-host properties that span multiple hosts and protocol steps—were left to the developer's intuition. For example, Kondo's proof of Paxos [19] required 20 such properties, which anecdotally took experts two person-weeks to derive.

In this work, we set our sights on a more ambitious goal of *automatically finding inductive invariants in an undecidable setting*. Of course, by the very nature of undecidability, this is an impossible task to achieve for all protocols. And yet, with the right insights, we believe this problem can be solved for many practical, complex protocols, much like how modern SAT solvers can solve large practical instances in reasonable time, despite SAT being an NP-hard problem.

We achieve this ambition based on two insights, which together drastically increase the amount of automation that we can achieve in finding inductive invariants.

Our first insight is that inter-host properties, which prior work delegated to the developer's intuition [11, 40], can be proven as a consequence of multiple simpler invariants that relate host states to network messages. For example, if host R's current state is $x = 5$, that can be traced back to the protocol step that most recently updated x based on some received message M, usually based on some field of M. Message M can in turn be traced back to the state of the message's sender, S, creating a causal chain between the hosts R and S that is invariant. We introduce *Provenance Invariants* as a way to keep track of this lineage. Provenance Invariants are always derived by statically inspecting the steps of single hosts in isolation, which makes them much more intuitive and frees the developer from deriving complex, inter-host properties.

The use of Provenance Invariants, however, creates a new challenge, as we now have a much larger number of simple invariants to generate. Recent prior work [40] automatically generates only one of the three categories of such invariants: those related to steps that send network messages. The other two categories, namely, invariants related to receiving messages and local steps, were either only partially automated or not supported at all.

Our second insight allows us to significantly automate the tracking of provenance through the system's execution. We observe that if a part of a host's state, which we call a *shard*, is always updated atomically—no steps modify only some variables in the shard but not others—then we can automatically track the provenance of this shard to one of the steps that modify it. Concretely, if a shard is in a non-initial state, it means that one of the steps that modify it must have occurred. We propose an *atomic sharding algorithm* to determine such shards and generate Provenance Invariants from them.

Using these insights, we developed Basilisk, a tool that uses Provenance Invariants and atomic sharding to automatically derive inductive invariants and prove their inductiveness for a wide variety of distributed protocols, with the help of minor hints from the developer. In doing so, Basilisk relieves developers from manually formulating an invariant and proving its inductiveness. This allows the developer to focus their efforts on the final task, using the Basilisk-generated inductive invariant to prove the desired safety property. This remaining proof is conceptually *significantly easier*, as it does not involve identifying an inductive invariant.

We evaluate Basilisk's utility by applying it to 16 protocols, including notoriously complex ones like Multi-Paxos [19]. Our evaluation shows that in all cases, the inductive invariants of these protocols can be expressed using simple Provenance Invariants that can be derived from a single host's steps. We further show that Basilisk can find all the inductive invariants for all 16 protocols almost fully automatically.

In summary, this paper makes the following contributions.

- It advances the state-of-the-art in automatic invariant inference by structuring inductive invariants as a collection of simple invariants that can be derived from the protocol steps of individual hosts.
- It introduces *atomic sharding*, a technique that allows us to automatically track the provenance of host states through the protocol's execution.
- It introduces Basilisk, a tool that uses the above techniques to automatically derive the inductive invariants of a wide range of distributed protocols, with only occasional minor hints from the developer.

2 The Burden of Inductive Invariants

Proving that a distributed protocol is safe entails defining a desirable safety property φ (e.g., that all nodes reaching a decision agree on the outcome) and then showing that φ is an *invariant* (i.e., that φ holds in all reachable states of the protocol's execution).

In all but the most trivial protocols, however, φ is not an *inductive* property, even if it is indeed an invariant. This means that there exist (unreachable) states satisfying φ that can transition to an unsafe state. Because it is not possible to decide whether a given state is reachable, proving φ's invariance requires identifying a stronger property representing an over-approximation of the set of reachable states—an *inductive invariant* I—that (1) implies φ, (2) holds in the initial state, and (3) is *inductive*, meaning that if I holds in some state, it holds after the protocol takes one step from that state. Typically, I is expressed as the conjunction of a series of smaller invariants, $I = I_1 \wedge \cdots \wedge I_m$.

According to conventional wisdom, if we reject the onerous restrictions of decidable logics (§1), then finding an inductive invariant is a creative and laborious endeavor. Even after designing the protocol, the developer typically does not know all of the clauses needed to construct an inductive invariant. Instead, the developer begins with an educated guess of a candidate invariant, and then follows an iterative *invariant-proof*

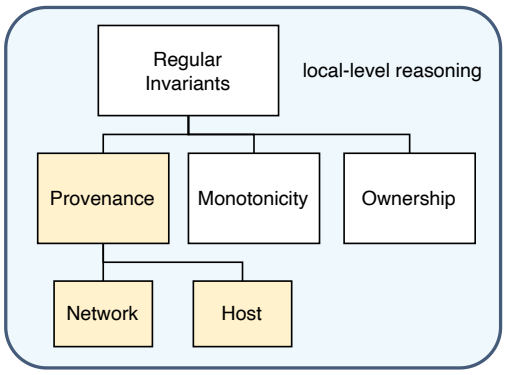


Figure 1: Our invariant taxonomy. We introduce the concept of Provenance Invariants, while Monotonicity Invariants and Ownership Invariants are originally introduced in Kondo [40].

loop: they attempt to prove that the candidate invariant is inductive, fail to complete the proof, and then refine the candidate based on feedback gained from the proof attempt. Each iteration involves writing proof code and creatively devising changes to the candidate invariant. Unfortunately, it might take many iterations before the developer arrives at a correct inductive invariant and proof. Indeed, in frameworks such as IronFleet [10] that require the developer to manually devise and prove inductive invariants, such a process reportedly took months for complex protocols like Multi-Paxos.

Recently, Kondo [40] took a first step towards automating this burdensome invariant-proof loop. It proposes an invariant taxonomy that carves out a category of invariants that can be systematically derived, so that only the remainder need to be hand-written by the user. Kondo observes that these manually written invariants often talk only about host state and not the network, but this simplification does not spare the user from going through the invariant-proof loop. Coming up with the required properties often requires a deep understanding of why the protocol works, since the properties span multiple hosts and steps, and cannot be easily derived from the protocol description. This understanding can be difficult to translate to an invariant even for the protocol's designer, let alone a verification engineer working with an existing protocol.

3 Expressing Inductive Invariants with Only Regular Invariants

We present how we use only simple, mechanically derivable invariants—called Regular Invariants—to craft inductive invariants that prove the safety of distributed protocols.

Regular Invariants (Figure 1) are a class of invariants introduced in Kondo's invariant taxonomy [40]. The defining feature of Regular Invariants is that they capture localized properties, which can be easily derived by examining steps of individual hosts in the protocol, without concern for how hosts interact. As a result, they are both easy for developers

```
1:  datatype Preference = Yes | No
2:  datatype Decision = Abort | Commit
3:  datatype Message =
4:      VOTE(v: Preference, src: nat) | DECIDE(d: Decision)
5:  datatype Option⟨T⟩ = None | Some(v: T)
6:  datatype Coordinator = Variables(
7:      numParticipants: nat,                    // some constant N
8:      decision: MonotonicOption⟨Decision⟩,     // initially None
9:      yesVotes: MonotonicSet⟨nat⟩,             // initially empty
10:     noVotes: MonotonicSet⟨nat⟩               // initially empty
11: )
12: datatype Participant = Variables(
13:     hostId: nat,                             // unique identifier ∈ [0,N)
14:     preference: Preference,                  // non-deterministic constant
15:     decision: MonotonicOption⟨Decision⟩      // initially None
16: )
```

Figure 2: Hosts and message states of the Two-Phase Commit protocol. Since there is only one coordinator, the coordinator's identifier is implicit. Moreover, since VOTE messages are destined for the coordinator, and DECIDE is broadcast to every participant, we omit their destination fields. MonotonicSets are add-only, while MonotonicOptions are write-once.

to derive manually and well-suited to automation. As a subclass of Regular Invariants, Monotonicity Invariants capture the "add-only" nature of data types such as monotonic counters, grow-only sets, and append-only lists, by asserting how their values may evolve during an execution. Another subclass is Ownership Invariants, which describe the ownership of unique resources in the system. We use these subclasses unchanged from Kondo's invariant taxonomy.

Our key insight is that a new class of Regular Invariants, which we call *Provenance Invariants* (§3.2), can be used to express both *inter-host* (§3.3) and *intra-host* (§3.4) properties in distributed protocols. These properties were previously beyond the scope of Regular Invariants and automation tools [40], leaving developers to manually derive and prove them. With the introduction of Provenance Invariants, we can express these properties, and in turn entire inductive invariants of distributed protocols, using only Regular Invariants. We present our algorithm to automatically derive Provenance Invariants in §4.

To ground our discussion, we use the classic Two-Phase Commit protocol as a running example in the rest of the paper.

3.1 Running Example: Two-Phase Commit

In this protocol, a group of participants work with a single coordinator to decide whether to abort or commit a transaction. Figure 2 specifies the states of hosts in the system.

The protocol runs as follows, with Figure 3 formally defining some of these steps:

```
1:  // Local computation that doesn't receive or send messages
2:  step CoordinatorMakeDecision(
3:      v: Coordinator, v′: Coordinator)
4:      ∧  v.decision = None
5:      ∧  if |v.yesVotes| = v.numParticipants then
6:             v′ = v.(decision := Some(Commit))
7:         else if |v.noVotes| > 0 then
8:             v′ = v.(decision := Some(Abort))
9:         else
10:            v′ = v
11:
12: step CoordinatorSendDecide(
13:     v: Coordinator, v′: Coordinator, send: Message)
14:     ∧  v.decision.Some?
15:     ∧  send = DECIDE(v.decision.value)
16:
17: step ParticipantReceiveDecision(
18:     v: Participant, v′: Participant, recv: Message)
19:     ∧  recv.DECIDE?
20:     ∧  v.decision = None
21:     ∧  v′ = v.(decision := Some(recv.d))
22:
```

Figure 3: Some examples of the hosts' steps in the Two-Phase Commit protocol. Each step defines a relation between a host's current state v and its new state v' after taking the step. Expressions of the form $v' = v.(X := Z)$ indicates that v' is identical to v except for the field $v'.X$, which has the value Z. Note that we use the conjunction operator $\wedge$ to denote a bulleted list of conjuncts, and the '?' syntax is used to assert if a value is of a particular type or variant.

*a*1. A single coordinator and N participants are initialized.

*a*2. Participants send their preferences to the coordinator through VOTE messages.

*a*3. Upon receiving a VOTE message, the coordinator updates its *yesVotes* or *noVotes* appropriately.

*a*4. If the coordinator has obtained Yes votes from every participant, it decides Commit. Otherwise, if it received a No vote, it decides Abort (*CoordinatorMakeDecision* step in Figure 3).

*a*5. Once the coordinator makes a decision, it broadcasts it as a DECIDE message (*CoordinatorSendDecide*).

*a*6. A participant, on receiving a DECIDE message, sets its local *decision* to match (*ParticipantReceiveDecision*).

The safety property we want to prove is that if any participant decides Commit, then every participant's local *preference* must be Yes.

3.2 Provenance Invariants and Execution Histories

Provenance Invariants are expressed using the notion of *history-preservation* [40]. A history-preserving protocol model augments a host's current state with an append-only log of previous states, referred to as its *history*. This history captures a snapshot of every state a host transitions through, from its initial state to its current state.

Formally, a Provenance Invariant connects a property of the current state of the system to how an adjacent pair of states in a host's history must be related by the host taking some step, where this step is one that made the property true. We define two categories of Provenance Invariants.

First, a *Network-Provenance Invariant* relates a message m in the network to the execution of one of a few host steps $T_1, \ldots, T_j$ that must have sent it. Formally:

$$\begin{aligned} m \in network \implies \exists\, i : & \; T_1(hist[i], hist[i+1], m) \\ & \vee \ \ldots \\ & \vee \ T_j(hist[i], hist[i+1], m) \end{aligned}$$

where *hist* is the history sequence of m's sender. This disjunction of steps is necessary because it is not always possible to attribute the provenance of a message to a unique step, such as when multiple steps send the same message.

In our Two-Phase Commit example, the following statement is a Network-Provenance Invariant:

> "For every DECIDE message m in the network, there must be adjacent states v, v' in the coordinator's history such that $CoordinatorSendDecide(v, v', m)$."
> (*Decide-Msg-Provenance*)

Second, a *Host-Provenance Invariant* relates a local property q of a host's current state h_{cur}—which we call a *provenance witness*—to one of a few steps the host must have taken in order for the local property to be satisfied. This witness q must be a property that is not true in any initial states of the host. Formally, with *hist* as the history sequence of the host in question:

$$\begin{aligned} q(h_{cur}) \wedge \big(\forall\, h : \neg\,(HostInit(h) \wedge q(h))\big) \implies \exists\, i : & \\ \neg\, q(hist[i]) \wedge q(hist[i+1]) & \\ \wedge\ (T_1(hist[i], hist[i+1]) & \\ \vee \ \ldots & \\ \vee \ T_j(hist[i], hist[i+1])) & \end{aligned}$$

Again, we have a disjunction of steps because it is possible that there is more than one step that makes q true.

For example, the following is a Host-Provenance Invariant in Two-Phase Commit, with its antecedent serving as the

provenance witness:

> "If a participant's *decision* is Commit, there must be adjacent states v, v' in its history, together with a received message m, such that *ParticipantReceiveDecision*(v, v', m)."
> (*Participant-Decision-Provenance*)

Crucially, every individual Provenance Invariant is inductive, due to the immutable nature of execution histories. For instance, consider the property *Decide-Msg-Provenance* above. It is trivially inductive because for any message in the network, the fact that the sender performed a step that sent it is forever recorded into the history, so there is no subsequent transition that ever can make this invariant false. We leverage this feature to automatically generate inductive invariants of distributed protocols (§5.3).

3.3 Expressing Inter-Host Relationships Using Provenance Invariants

We demonstrate that global relationships between remote hosts—which prior work [10, 40] asked the developer to creatively supply and manually prove—can actually be expressed as conjunctions of well-chosen Host-Provenance Invariants and Network-Provenance Invariants.

To illustrate, consider the inter-host property:

> "If some participant decided Commit, then the coordinator also decided Commit."
> (*Participant-Agreement*)

This property is a clause in the inductive invariant of Two-Phase Commit that Kondo [40] required developers to craft manually.

Using Provenance Invariants, we can imply *Participant-Agreement* by the conjunction of *Participant-Decision-Provenance* and *Decide-Msg-Provenance*. In particular, suppose that a participant host h decided Commit. Then by *Participant-Decision-Provenance*, h must have executed step *ParticipantReceiveDecision*. By the body of this step (Figure 3), we know that h received a message $m = \text{DECIDE}(\text{Commit})$ from the network. Next, we infer from *Decide-Msg-Provenance* that m's sender (namely, the coordinator), must have sent m via step *CoordinatorSendDecide*. This step itself says the coordinator's decision agrees with the message at the time of sending. Given that the coordinator does not equivocate on its decision, we arrive at our conclusion that the coordinator's current decision is also Commit.

More formally, when a host r updates its state upon receiving a message m, we can trace this update to the execution of some step T_r via a Host-Provenance Invariant. This induces a relationship $\beta(r, m)$ between the updated host state and m, given directly by T_r's specification. In addition, the receipt of m implies, via a Network-Provenance Invariant, that m's sender s executed a step T_s that sent m. This creates another relationship $\alpha(s, m)$ relating the host s and the message m, given by T_s's specification. When taken together, α and β imply some inter-host relationship between s and r, namely, $\exists m : \alpha(s, m) \wedge \beta(m, r)$. By transitively chaining such arguments, we can derive relationships between the head and tail hosts of a causal chain of messages.

The above argument establishes how inter-host properties, which logically connect the state of recipient hosts to the state of sender hosts, may be implied using Provenance Invariants. With this technique, inductive invariants do not need to explicitly describe inter-host properties, which prior work like IronFleet and Kondo required users to creatively and laboriously produce. Instead, they are replaced with simpler, localized Provenance Invariants.

3.4 Expressing Local Properties Using Provenance Invariants

Beyond inter-host properties, an inductive invariant may also describe local properties of individual hosts. One example is:

> "If the coordinator decided Commit, then its *yesVotes* set is of size *numParticipants*."
> (*Coordinator-Decision-Provenance*)

Prior work [10, 40] required the user to manually derive such properties. We capture them with ease using Host-Provenance Invariants, which lend themselves to automation (§4).

Our observation is that local properties such as *Coordinator-Decision-Provenance* can be attributed to local computation steps, i.e., steps that do not receive a message. Examples of such steps include those that are triggered by a timeout or local conditions.

Just as a Host-Provenance Invariant may trace the provenance of a state update at a host to a step that receives a message, it may likewise trace it to a local computation step. For instance, to derive *Coordinator-Decision-Provenance*, the Host-Provenance Invariant:

> "If the coordinator decided Commit, then it must have executed step *CoordinatorMakeDecision*."

attributes the coordinator's Commit decision to step *CoordinatorMakeDecision*. This step itself says that for the coordinator to decide Commit, it must have satisfied in a previous state the enabling condition that it has gathered Yes votes from every participant. Then, given the monotonic add-only nature of the *yesVotes* set, we conclude that *yesVotes* set is of the threshold size.

The next section presents our technique for automatically deriving the Provenance Invariants that underpin our proofs.

4 Automating Provenance Invariants With Atomic Sharding

To get the most out of expressing inductive invariants using only Regular Invariants, we aim to derive the required Provenance Invariants automatically given a distributed protocol. We first explain a new *atomic shard principle* to describe a category of mechanically derivable Host-Provenance Invariants (§4.1). We then present an *atomic sharding algorithm* to automatically derive invariants based on this principle (§4.2). Finally, we explain how we derive Network-Provenance Invariants (§4.3).

4.1 Automating Host-Provenance with the Atomic Shard Principle

To automatically derive Host-Provenance Invariants, we first identify a set of useful provenance witnesses. For each witness q, we then determine the set of candidate steps that could affect its provenance. Whenever q holds, we know that at least one of these steps must have been executed.

We propose the atomic shard principle to meet this goal. First, we define a *shard* to be a subset of a host's local variables. A shard σ is *atomic* if all variables in the shard are always updated "atomically"—there are no protocol steps that only modify some variables within σ but not others. Each atomic shard σ is then associated with the set of steps A_σ that modify its variables.

The *atomic shard principle* states that when the current value of some variable in an atomic shard σ differs from its initial value, then every variable in this shard must have attained their current values through an atomic step in A_σ.

From this principle, we obtain one Host-Provenance Invariant per atomic shard. Formally, let $\sigma := \{x_1, \ldots, x_k\}$ be an atomic shard of a host, and suppose $A_\sigma := \{T_1, \ldots, T_j\}$ are the protocol steps that modify this shard. Then, given that h_{cur} is the host's current state, we first define a provenance witness as the predicate:

$$q(h) := (h.x_1 = h_{\text{cur}}.x_1) \wedge \ldots \wedge (h.x_k = h_{\text{cur}}.x_k)$$

The atomic shard principle tells us that if the host's current state satisfies q, and if q is not true in any initial states of the host, then variables x_1 through x_k must have attained their current values when the host executed one of the steps in A_σ at some point in its history. This is the Host-Provenance Invariant:

$$\begin{aligned} & q(h_{cur}) \wedge \big(\forall\, h : \neg(\mathit{HostInit}(h) \wedge q(h))\big) \implies \exists\, i : \\ & \quad \neg\, q(\mathit{hist}[i]) \wedge q(\mathit{hist}[i+1]) \\ & \quad \wedge\ (T_1(\mathit{hist}[i], \mathit{hist}[i+1]) \\ & \qquad \vee \ldots \\ & \qquad \vee\ T_j(\mathit{hist}[i], \mathit{hist}[i+1])) \end{aligned}$$

where *hist* is the history sequence of the host.

As a concrete example, consider the Host-Provenance Invariant *Participant-Decision-Provenance* from §3.2. We observe that $\{decision\}$ is an atomic shard at the participant host, and *ParticipantReceiveDecision* is the only step that modifies this shard. From this, we infer *Participant-Decision-Provenance*:

$$\begin{aligned} & h_{cur}.\text{decision} = \text{Some}(\text{Commit}) \implies \exists\, i : \\ & \quad \mathit{hist}[i].\text{decision} \neq \text{Some}(\text{Commit}) \\ & \quad \wedge\ \mathit{hist}[i+1].\text{decision} = \text{Some}(\text{Commit}) \\ & \quad \wedge\ \mathit{ParticipantReceiveDecision}(\mathit{hist}[i], \mathit{hist}[i+1]) \end{aligned}$$

4.2 Atomic Sharding Algorithm

We present an *atomic sharding algorithm* to automatically identify atomic shards σ and their associated steps A_σ. This occurs in three phases. First, we estimate the *footprint* of each host step. Second, we analyze how footprints intersect to identify atomic shards. Finally, we further refine each shard to more effectively handle collection-type variables such as sets and maps.

Estimating footprints. We define a host step's *footprint* to be a set of local variables that the step may modify. For each step, we estimate its footprint via a static analysis of the host's description. §5.3 describes how we implement this analysis in practice.

Importantly, it is safe to overestimate a step's footprint, which our syntax-based approach inevitably does—a footprint may contain variables that the step never modifies, or variables that may only be conditionally updated. Too much overestimation, however, leads to weaker and less useful invariants. On the other hand, our algorithm never underestimates footprints, which would result in incorrect invariants.

Our algorithm (and hence the Basilisk tool—§5) currently rejects a protocol if it contains a footprint with two variables that are conditionally updated via different conditions. For example, it prohibits a step that either updates x or y with the expression "if c then $x := 5$ else $y := 6$". This restriction does not sacrifice generality, as any such step may be equivalently written as two steps, with the enabling condition c for a step that updates x and $\neg c$ for another that updates y, respectively. We currently rely on the developer to perform such transformations wherever they are needed.

Computing maximal atomic shards. Next, we identify atomic shards by analyzing footprint intersections to find subsets of host variables that are always updated atomically.

This extraction process is best visualized using a Venn diagram. Let steps S_1, S_2 and S_3 be the three steps that a hypothetical host takes. The ovals F_1, F_2 and F_3 in Figure 4 represent the respective footprints of these steps. By definition, each partition in the Venn diagram is an atomic shard:

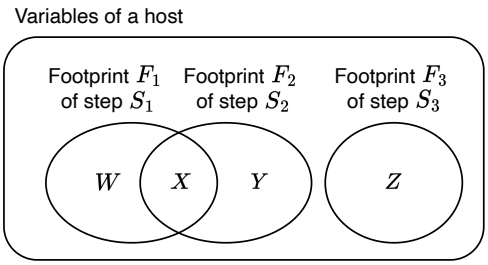


Figure 4: The footprints of three steps of a hypothetical host. Regions W through Z represent maximal atomic shards.

- region $W \coloneqq F_1 \setminus F_2$, with associated steps $A_W \coloneqq \{S_1\}$,
- region $X \coloneqq F_1 \cap F_2$, with $A_X \coloneqq \{S_1, S_2\}$,
- region $Y \coloneqq F_2 \setminus F_1$, with $A_Y \coloneqq \{S_2\}$, and
- region $Z \coloneqq F_3$, with $A_Z \coloneqq \{S_3\}$.

This process extracts *maximal* atomic shards, which are ones where adding an additional variable to it would violate its atomicity. In most cases, maximal atomic shards provide greater utility over ones that are non-maximal. To illustrate, consider a host with two local variables x and y, and $\{x, y\}$ is a maximal atomic shard modified by a single step T. Suppose T receives a message m and sets $x \coloneqq m.x$ and $y \coloneqq m.y$. The atomic shard principle yields the Host-Provenance Invariant:

> "If x or y do not hold their initial values, then they must have obtained their current values through a message m, where $m.x = x$ and $m.y = y$."

However, if we split $\{x, y\}$ into non-maximal atomic shards $\{x\}$ and $\{y\}$, we obtain two invariants:

- If x does not hold its initial value, it must have obtained its current value through a message m_1, where $m_1.x = x$; and,
- if y does not hold its initial value, it must have obtained its current value through a message m_2, where $m_2.y = y$.

This is strictly weaker than the single invariant we draw from the maximal shard. For instance, we can no longer prove the existence of a singular message m that satisfies both $m.x = x$ and $m.y = y$.

Additional refined atomic shards. In the special case when shards contain collection-type variables (e.g., sets and maps), however, maximal atomic shards yield limited benefit. This is because these collection types "collapse" provenance, making it difficult to relate them to other fields.

To demonstrate, consider a host with two local variables x: int and s: set⟨int⟩, where the latter is initially empty, and suppose $\{x, s\}$ is a maximal atomic shard that is modified by a single step T. Further suppose that T receives a message m and sets $x \coloneqq m.x$ and adds $m.y$ to s. Using the $\{x, s\}$ shard, we infer the Host-Provenance Invariant:

> "If x or s do not hold their initial values, then they must have obtained their current values through T."

However, this is not strong enough for us to trace the provenance of the host's current state to the message that led to that state, which is the primary benefit of a Host-Provenance Invariant. In particular, we cannot say:

> "For all items e in s, if e is not initially in s or x does not hold its initial value, then there must be a message m with $m.x = x$ and $m.y = e$."

This property is false because e may have arrived in a message $\hat{m}$ with $\hat{m}.x = \hat{x} \neq x$, and x was subsequently updated to its current value by a completely different message.

Hence, to trace the provenance of specific items in collections such as s, we further identify *refined* atomic shards that package every collection-type variable in its dedicated (non-maximal) shard.

With refined shard $\{s\}$, we can reason about the step that introduced each individual item in s:

> "For all items e in s, if e is not initially in s, then e must have been added to s through T."

which establishes the additional fact that the host received a message m where $m.y = e$ for every e in s.

Finally, note that while atomic sharding always generates correct invariants, it does not cover the space of all Host-Provenance Invariants. There are instances where it may fail to capture necessary properties, and hence the developer must supply an additional hint. We describe these hints in §5.4 and characterize the scenarios requiring them in §6.4. However, our evaluation shows that such cases are rare in practice.

4.3 Automating Network-Provenance

The procedure for generating Network-Provenance Invariants is much simpler. For each message variant M defined in the protocol, we identify a set of steps $\{T_1, \ldots, T_j\}$ that send messages of variant M. We then infer the following Network-Provenance Invariant:

$$\begin{aligned} \forall\, m :\ & \text{variant}(m) = M \ \wedge\ m \in S.\text{network} \\ \implies & \exists\, i : \big(T_1(hist[i], hist[i+1], m) \\ & \qquad \vee\ \ldots \\ & \qquad \vee\ T_j(hist[i], hist[i+1], m)\big) \end{aligned}$$

where *hist* is the history sequence of m's sender.

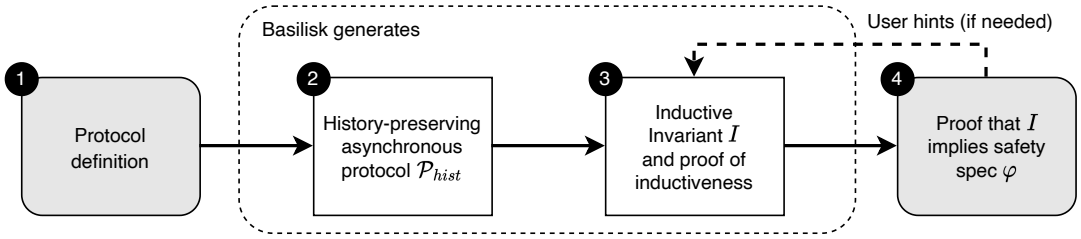


Figure 5: Basilisk workflow. Shaded bubbles represent artifacts that the user writes, while white boxes represent artifacts that Basilisk generates. In a few cases, the user may discover while writing the proof in step ❹ that the inductive invariant I generated in ❸ is too weak. In such cases, the user provides hints (the dashed arrow) that enable Basilisk to generate a stronger invariant.

5 Proving Protocols With Basilisk

Armed with the power and simplicity of Provenance Invariants, we design and implement Basilisk, a tool that finds the inductive invariants of distributed protocols with minimal guidance from the user.

Basilisk is designed for protocol models written in general verification frameworks such as Dafny [20], where determining the inductiveness of an invariant is an undecidable problem. It targets asynchronous message passing systems in which hosts communicate over an unreliable network that may arbitrarily delay, drop, duplicate, or re-order messages. We choose this network model for its generality and prevalence in real systems. While we do not explore stronger models in this work, we expect that their stronger guarantees would enable us to generate stronger invariants.

We implement the Basilisk prototype by modifying the Kondo codebase [39], which itself extends the Dafny version 4.2 verifier [1]. Overall, Basilisk's implementation adds about 2,000 lines of C# code to the Dafny verifier.

Figure 5 illustrates the Basilisk developer workflow. Step ❶: The developer starts by defining the hosts in the protocol, including their initialization conditions and local state transitions. Step ❷: Basilisk automatically generates a history-preserving asynchronous protocol $\mathcal{P}_h$. It models an asynchronous network and maintains the execution history of hosts. Step ❸: Basilisk generates an inductive invariant of $\mathcal{P}_h$, along with a proof of its inductiveness. Step ❹: The user proves that this inductive invariant implies the protocol's safety property. In rare instances, to complete this proof, the user may need to provide hints so that Basilisk will produce a stronger inductive invariant.

5.1 Step ❶: Protocol Definition

The user defines the state and protocol steps of each type of host in the system (e.g., Figures 2 and 3). Each step may send a message (including broadcast), receive a message, perform both send and receive, or do neither. This is a more permissive model than Kondo, which prohibits steps that both send and receive. Basilisk extends easily to steps that send or broadcast more than one message; we did not implement that as none of the protocols in our evaluation required such a feature.

The user then defines a Hosts record representing the state of all hosts in the system. Each field in the record is a list of hosts of a particular type. For example, in Two-Phase Commit:

$$\textbf{datatype}\ \text{Hosts} = \text{Hosts}(\textit{coordinators}\text{: seq}\langle\text{Coordinator}\rangle,\ \textit{participants}\text{: seq}\langle\text{Participant}\rangle)$$

The user then describes the initial states of Hosts using a *HostsInit* predicate. Typically, this specifies that each host satisfies its own local initial conditions, together with some global state constraints. For instance, in Two-Phase Commit:

$$\begin{aligned}&\textbf{predicate}\ \text{HostsInit}(s\text{: Hosts})\\ &\quad\wedge\ |s.\text{coordinators}| = 1\\ &\quad\wedge\ \textit{CoordinatorInit}(s.\text{coordinators}[0],\ |s.\text{participants}|)\\ &\quad\wedge\ \forall\, id :\ \big(0 \le id < |s.\text{participants}| \implies \textit{ParticipantInit}(s.\text{participants}[id], id)\big)\end{aligned}$$

where *CoordinatorInit* and *ParticipantInit* circumscribe the initial states of the coordinator and participant hosts.

Finally, the user defines a state transition relation for each host in the system. This is a disjunction of the possible steps that the host can take. For example, the coordinator in Two-Phase Commit has the transition relation:

$$\begin{aligned}&\textbf{predicate}\ \text{CoordinatorNext}(h\text{: Coordinator},\ h'\text{: Coordinator},\ \textit{send}\text{: Message},\ \textit{recv}\text{: Message})\\ &\quad\vee\ \ldots\\ &\quad\vee\ \textit{CoordinatorSendDecide}(h, h', \textit{send})\\ &\quad\vee\ \textit{CoordinatorMakeDecision}(h, h')\end{aligned}$$

where *CoordinatorSendDecide* and *CoordinatorMakeDecision* are defined in Figure 3.

Monotonicity hints. Like in Kondo [40, §3.1], the user labels monotonic variables, such as grow-only sets, when specifying hosts. These types implement a *less-than-or-equal-to (lteq)* partial order relation that captures their monotonic properties, which Basilisk uses to express Monotonicity Invariants in step ❸.

5.2 Step ❷: Asynchronous Protocol Model

With the above inputs, Basilisk generates a *history-preserving* asynchronous protocol $\mathcal{P}_h$. This model includes a sequence *history* where each entry is a snapshot of all hosts in the system, with the latest entry representing their current state. The global state of the protocol contains this history, together with an asynchronous network that is modeled as a monotonically increasing set of sent messages:

datatype GlobalState = GS(*history*: seq⟨Hosts⟩,
network: set⟨Message⟩)

Similar to prior work [10, 40], the protocol's behavior is expressed using the temporal logic of actions (TLA [17]) as a state machine that can perform a non-deterministic series of atomic steps. At first, *history* holds only the initial state of the system, and the network has no messages:

predicate Init(S: GlobalState)
$\wedge$ $|S.\text{history}| = 1$
$\wedge$ HostsInit(S.history[0])
$\wedge$ $S.\text{network} = \emptyset$

In each transition of the system, a non-deterministically chosen host performs a non-deterministically chosen local step. If this step involves the host receiving a message, then the message must be from the *network* set and addressed to the receiver host. If the step involves sending a message, the message is added to the *network* set, with the sender host marked as its source. Moreover, the new Hosts snapshot is appended to the *history* sequence:

step Next(S: GlobalState, S': GlobalState)
$\wedge$ $|S.\text{history}| \geq 1$
$\wedge$ $S.\text{history} = \mathit{trunc}(S'.\text{history})$
$\wedge$ SystemNext($\mathit{last}(S.\text{history})$, $S.\text{network}$,
$\mathit{last}(S'.\text{history})$, $S'.\text{network}$)

Here, given a sequence s, $\mathit{trunc}(s)$ yields s with the last item removed, while $\mathit{last}(s)$ returns the last item. *SystemNext* describes how the current state of hosts and the network changes as the result of the step, calling upon the host transition relations the user defined in step ❶.

5.3 Step ❸: Generating Regular Invariants

From $\mathcal{P}_h$, Basilisk automatically generates an inductive invariant I. This I is expressed as the conjunction of a series of Regular Invariants. This includes Provenance Invariants and Monotonicity Invariants. There is a final category, Ownership Invariants, which we inherit from Kondo [40, §3.1], but omit from this discussion as it is not the focus of this work.

Basilisk automatically generates Provenance Invariants using the atomicity principle described in §4. To compute footprints, Basilisk relies on parsing Dafny's state update syntax.

There are two formats with which a host step may modify local state. The first uses Dafny's state update syntax $v' = v.(Z_1 := X_1, \ldots, Z_k := X_k)$. This expression means that host state v' is identical to current state v except for the fields Z_1 through Z_k, which are assigned the values X_1 through X_k, respectively. The second format expresses some relation $r(v'.Z, v, m)$ between an updated field $v'.Z$, the current host state v, and any incoming message m. For simplicity, the Basilisk prototype permits only the former update syntax.

Meanwhile, Basilisk inherits Kondo's technique for generating Monotonicity Invariants. For each monotonic variable specified in step ❶, Basilisk generates an invariant stating that the variable satisfies its *lteq* relation as its value evolves over history. Such invariants assert the monotonic nature of their data types (e.g., the coordinator host's non-equivocation of its decision when proving *Participant-Agreement* in §3.3), and are crucial in stating how some properties that are true of a host's past state are also true in its present state.

Finally, Basilisk also produces a mechanically checked proof of I's correctness. Namely, given a pair of global states S and S', the proof shows that Init(S) implies $I(S)$ and that Next(S, S') $\wedge$ $I(S)$ implies $I(S')$. Given the mechanical and local nature of Regular Invariants, and the fact that each individual Provenance Invariant is inductive (§3.2), these proofs follow a systematic structure and are easy to formulate.

5.4 Step ❹: Proving Safety

Finally, let $\varphi(h\text{: Hosts})$ be the protocol's safety property. The user employs the inductive invariant I generated in the previous step to prove that $\varphi_h(S) := \varphi(\mathit{last}(S.\text{history}))$ is an invariant in $\mathcal{P}_h$. This involves two proof obligations proven as lemmas in Dafny, to show that $I(S) \wedge \varphi_h(S)$ is an inductive invariant. They are:

*Ob*1. Init(S) implies $I(S) \wedge \varphi_h(S)$; and,

*Ob*2. $I(S) \wedge \varphi_h(S) \wedge \text{Next}(S, S')$ implies $I(S') \wedge \varphi_h(S')$.

This procedure requires some creativity from the user, to show how a collection of simple Regular Invariants imply that, say, all hosts agree on the decision value in a consensus protocol. It is, however, a much more straightforward and well-defined task compared to IronFleet or Kondo. Here, the user concentrates only on proving safety using I, which is already inductive, and avoids going through the invariant-proof loop (§2) to discover an inductive invariant.

Possible failure modes. When trying to prove the above obligations, the user may fail in two ways. First, although Basilisk only generates correct inductive invariants, they may not be strong enough to establish safety. Our evaluation (§6.4) shows that this is usually because the atomic sharding algorithm cannot generate every correct Host-Provenance Invariant. As a remedy, the user manually supplies the provenance

witness for a desired Provenance Invariant as a hint to Basilisk, explaining how some particular host transitions affect local state (represented by the dashed arrow in Figure 5). Such hints are usually quite straightforward to provide, requiring far less effort than global, inter-host invariants.

Second, the protocol definition written in step ❶ may be incorrect, meaning it contains a step that violates safety.

Basilisk handles both failure modes gracefully. With Basilisk, the user proves safety in a modular fashion by showing that it holds across each protocol transition. As a result, when verification fails, the verifier advises the user on the exact host step where the proof breaks down. This helps the user identify which invariants may need to be strengthened or where a potential protocol bug might reside.

6 Evaluation

To assess the Basilisk methodology, we apply it to a diverse set of distributed protocols, listed in Table 1. This list is informed by our experience in designing and implementing distributed protocols, and seeks to cover the space of common protocols. We used the protocols in the first block in Table 1, namely Echo Server, Ring Leader Election, Simple Leader Election, Paxos, Paxos-Combined and Paxos-Dynamic, to develop and refine the concepts of Provenance Invariants and atomic sharding, before applying it to the remaining protocols in the second block. Notably, every protocol and its inductive invariant is outside of the EPR decidable fragment of logic [29].

We specify each protocol as a state machine in Dafny following the IronFleet style [10], together with a safety property (Appendix A contains a description of each protocol and its safety property). Using this state machine definition, Basilisk generates a provably correct inductive invariant I, expressed as a conjunction of Regular Invariants. Using I, we prove that the safety property is an invariant in the protocol, by manually proving the obligations $Ob1$ and $Ob2$ defined in §5.4.

We also compare Basilisk's results to the previous state-of-the-art, Kondo [40]. Note that because Kondo prohibits protocol steps that both receive and send messages, the protocol descriptions used by Kondo are modified from the Basilisk versions to accommodate this restriction. Because of the effort required to accommodate this restriction and prove these protocols using Kondo, we did not apply Kondo to every protocol.

Our evaluation determines whether Basilisk is effective in helping developers prove the safety of their distributed protocols. To do so, we address the following questions.

1. How effective is Basilisk in finding the inductive invariants of various distributed protocols? (§6.1)
2. How burdensome is expressing and proving protocols using Basilisk? (§6.2)
3. How efficiently can a verifier like Dafny verify the invariants and proofs produced using Basilisk? (§6.3)
4. What are the cases where the Basilisk methodology requires developer assistance? (§6.4)

6.1 Effectiveness of Basilisk in Finding Inductive Invariants

The *'User invs'* columns in Table 1 show the number of manually derived invariant clauses the user supplies to prove the safety of each protocol using Basilisk and Kondo, respectively. Basilisk succeeds in finding inductive invariants for all 16 distributed protocols on which it was evaluated. This is a significant improvement over Kondo, which required user-supplied invariants for most protocols.

Note that three ownership-based protocols (Distributed Lock, ShardedKV, and ShardedKV-Batched), each of which has a safety property that asserts the exclusive ownership of resources, can already be proven completely automatically using Kondo. Unsurprisingly, Basilisk achieves the same on these protocols, as Basilisk inherits the ownership reasoning of Kondo.

These results highlights the effectiveness of Basilisk along two fronts. First, it supports our hypothesis that many deep protocol properties can be implied by only simple Regular Invariants in our generalized invariant taxonomy. This includes properties of protocols that are widely regarded as very challenging for human developers, such as Multi-Paxos.

Second, it demonstrates the feasibility of finding the inductive invariants of distributed protocols automatically, even in general verification frameworks such as Dafny.

Altogether, Basilisk simplifies the developer's workflow, as manually finding inductive invariants demands a large amount of creativity and expertise. With Basilisk, users avoid the invariant-proof loop process of IronFleet and Kondo, where a developer must meticulously craft a candidate inductive invariant, only to realize that it is wrong after spending time to prove it correct.

6.2 Ease of Expressing and Proving Protocols

While Basilisk reduces the user's burden in finding inductive invariants, one might worry that it simply shifts this effort elsewhere rather than eliminates it, requiring the user to compensate with more labor in other aspects, such as writing more proof annotations. However, our evaluation demonstrates that Basilisk avoids this pitfall and materially improves the user's proof experience.

Protocol expressiveness. Given a conceptual protocol, expressing it in Basilisk is significantly easier than in Kondo, as evidenced by the fewer lines of code required to define the protocol (*'Size'* in Table 1). This is due to two reasons.

	Basilisk							Kondo						
	Size	User invs	Mono annots	Prov hints	Lines of proof *Safety*	Lines of proof *Total*	Time (sec)	Size	User invs	Mono annots	Recv hints	Lines of proof *Safety*	Lines of proof *Total*	Time (sec)
Echo Server [40]	157	0	1	0	8	49	5.1	260	1	1	0	0	40	7.9
Ring Leader Election [3]	99	0	0	0	80	115	5.3	179	1	0	0	30	63	8.0
Simple Leader Election [34]	149	0	1	0	45	87	7.3	255	3	1	0	33	94	13.6
Paxos [19]	418	0	5	0	444	486	24.6	631	20	5	2	558	777	42.9
Paxos-Combined	319	0	5	0	445	497	57.5	-	-	-	-	-	-	-
Paxos-Dynamic	414	0	4	2	479	522	42.4	-	-	-	-	-	-	-
Flexible Paxos [12]	418	0	5	0	441	483	22.8	633	20	5	2	559	780	49.4
Distributed Lock [10]	134	0	0	0	0	39	5.0	194	0	0	0	0	31	4.7
ShardedKV	152	0	0	0	0	39	5.9	213	0	0	0	35	68	5.0
ShardedKV-Batched	164	0	0	0	0	38	5.6	225	0	0	0	0	31	5.1
Lock Server [21]	212	0	0	0	0	38	7.3	287	1	0	0	20	59	6.3
Two-Phase Commit	278	0	3	0	95	139	7.4	385	4	3	0	119	186	8.9
Three-Phase Commit [32]	323	0	5	0	108	152	8.3	-	-	-	-	-	-	-
Reduce	135	0	0	0	30	69	24.4	-	-	-	-	-	-	-
Raft Leader Election [25]	172	0	1	2	52	93	6.3	-	-	-	-	-	-	-
Multi-Paxos [19]	447	0	4	2	522	565	61.5	-	-	-	-	-	-	-

Table 1: User experience metrics of Basilisk compared to Kondo. A lower number is better for every column, and the shaded cells under Kondo highlight areas where it is outperformed by Basilisk. *'Size'* counts the lines of code the user writes to express the protocol. *'User invs'* is the number of invariant clauses that the user manually supplies. *'Mono annots'* is the number of monotonic type annotations the user writes—these numbers are identical between Basilisk and Kondo. *'Lines of proof'* is the amount of proof code the user writes to complete the safety proof. In both Basilisk and Kondo, '*Safety*' is the size of the Dafny lemmas used to prove safety, while '*Total*' includes additional definitions of proof obligations. In Kondo, '*Total*' also includes the definitions of manually-written invariants, which Basilisk does not require. Finally, *'Time'* is the time Dafny takes to verify the final proof of the history-preserving asynchronous protocol.

First, in event-driven message passing systems, a common mode of operation for hosts is to receive a message, perform some computation based on the content of the message, and then send out messages in response. In Basilisk, such behavior is more naturally modeled as a single, atomic state machine step that receives the message, performs the computations, and sends the response (as is also the case in IronFleet [10]).

In contrast, because Kondo does not permit host steps that both receive and send messages, such host behavior must be modeled as two distinct steps—one that receives the message and performs the computation, and another that sends the responses. This leads to awkward protocol descriptions, as steps must be broken down into multiple smaller steps. For example, in Echo Server, the server host forwards a copy of every request it receives. In Kondo, this extremely simple step must be modeled as separate steps that a) receive and store the request locally, and b) send a response based on the locally stored request. Basilisk does not suffer from this restriction, and hence lets users specify their protocols more naturally.

Second, defining a protocol in Basilisk only requires the user to specify the initialization condition and state transition relation for hosts (§5.1). In contrast, Kondo additionally needs the user to define a synchronous version of the protocol [40, §4.2], resulting in more code and effort.

Proof annotation effort. To prove the correctness of a distributed protocol, both Basilisk and Kondo require the user to write additional proof code to prove that the desired safety property is indeed an invariant. We quantify this effort using the lines of code in the final proof (*'Lines of proof'* in Table 1).

For Basilisk, this effort involves writing a lemma to convince the Dafny verifier that Basilisk's generated inductive invariant implies the protocol's safety specification. Proving this lemma is the only step that demands user creativity when using Basilisk, and the size of this lemma is listed under the *'Safety'* sub-heading. Meanwhile, the *'Total'* sub-heading shows the total number of lines in the proof file, which includes the aforementioned lemma, and "standard lines" such as the definitions of the proof obligations *Ob*1 and *Ob*2, and import statements for other Dafny modules.

For Kondo, the *'Safety'* sub-heading counts lines of code in lemmas proving that the user-derived invariants, in conjunction with the safety property, is an inductive invariant in a synchronous version of the protocol [40, §4.2]. It also includes modifications that the user must then make to these

lemmas in order to prove the final asynchronous protocol. The *'Total'* count for Kondo includes the aforementioned lemmas and modifications, "standard lines" as described above, and also the definition of the user-supplied inductive invariant.

We observe that Basilisk is competitive with Kondo in terms of proof effort. For simple protocols such as Echo Server and Ring Leader Election, Kondo requires fewer proof annotations than Basilisk. However, Basilisk requires significantly less effort when handling complex protocols. In the case of Flexible Paxos, for example, Basilisk's safety lemma is 21% smaller than Kondo's (441 lines vs. 559 lines). Also, the user avoids defining their own inductive invariant, which comprises over 200 lines in Flexible Paxos. Importantly, this metric also hides the additional effort of the invariant-proof loop process demanded by Kondo, as it only shows the size of the final artifact, not the amount of proofs and invariants written and deleted while deriving the final product.

User guidance. Both Basilisk and Kondo rely on user guidance to automatically derive and prove invariants. Basilisk uses three types of hints: provenance witnesses, monotonicity labels and ownership labels.

Of these categories, the most interesting is Basilisk's use of provenance witnesses (listed under *'Prov hints'* in Table 1), which is a generalization of Kondo's receive witness conditions [40, §5.1] (listed under the *'Recv hints'*). Kondo required the user to provide one receive witness for every generated Receive Invariant, and it used Receive Invariants sparingly—two each for Paxos and Flexible Paxos (§7 explains the relationship between Host-Provenance Invariants and Kondo's Receive Invariants).

In comparison, Basilisk relies heavily on Host-Provenance Invariants, yet can generate most of them fully automatically through atomic sharding (§4). A hint is only required when the generated provenance invariants are not strong enough. For instance, only 6 out of 64 Host-Provenance Invariants used by the protocols in our evaluation required the user to provide provenance witnesses as hints. We characterize the conditions under which the user may need to manually provide provenance witnesses in §6.4.

Meanwhile, Basilisk requires the user to annotate monotonic data types and resource ownership semantics [40]. In these respects, Basilisk inherits the behavior of Kondo, resulting in an identical user experience. The *'Mono annots'* columns in Table 1 list the number of annotations used to prove each protocol; protocols that are available in Kondo share the same numbers as Basilisk. As for ownership labels, only Distributed Lock, ShardedKV, ShardedKV-Batched and Lock Server use them. We omit ownership from Table 1 because the user experience is identical to Kondo.

6.3 Verification Latency of Basilisk Invariants and Proofs

Verifier latency plays a crucial role in ensuring a positive developer experience, as it enables a fast feedback loop that helps users quickly complete or correct incomplete or erroneous proofs. The *'Time'* columns in Table 1 report the time taken by the Dafny verifier to check the final proof of each history-preserving asynchronous protocol. These times include checking the invariants and lemmas generated by the Basilisk and Kondo tools, as well as any user-supplied ones. Notably, they exclude the verification time of the synchronous protocol required by Kondo. The time required for both Basilisk and Kondo to produce their auto-generated invariants and lemmas is less than 2 seconds in all cases, and does not contribute significantly to overall verification latency.

Our results show that Basilisk achieves good verification performance that scales reasonably with protocol complexity. Even for challenging protocols like Multi-Paxos, verification completes in a minute, providing an efficient turnaround that supports interactive development.

Interestingly, although not a design goal, we observe that Basilisk outperforms Kondo in verification speed, particularly on complex protocols. For example, the Flexible Paxos proof verifies in less than half the time with Basilisk compared to Kondo. We hypothesize that this improvement stems from two reasons.

First, as explained in §6.2, the improved expressivity of Basilisk allows a protocol to be expressed with fewer host steps compared to Kondo. This reduces the complexity of the host state machines, resulting in a smaller burden on the verifier.

Second, Basilisk and Kondo differ in how proofs are authored and optimized. Dafny's verification time is sensitive to proof structure, and Basilisk users write the final asynchronous proof directly, allowing them to iteratively optimize it for fast verification. In contrast, Kondo users focus on writing a synchronous proof, and the asynchronous proof is generated from it without optimization for verifier performance.

6.4 Limitations

Atomic sharding algorithm. First, our atomic sharding algorithm does not identify every Provenance Invariant of a given protocol. In particular, it cannot derive relationships between variables when their relationships are only *implicitly* established across multiple steps.

As an example, consider a host that keeps track of all events occurring during the current calendar year. This set of events is emptied when the year changes. The host keeps the local variables *currentYear*: int and *events*: set⟨string⟩ that is initially empty. The host has two steps: T_1 receives a message m and adds $m.event$ to *events* only if $m.year = currentYear$; T_2 increments *currentYear* and resets $events := \emptyset$. In this case,

Basilisk generates the following invariant from observing T_1:

> "For all e in $h.events$, there is a message m and an h_{hist} in the host's history such that $m.event = e$, and $m.year = h_{hist}.currentYear$."

where h is the current state of the host. This invariant fails to imply the stronger property that for any such m, $m.year = h.currentYear = h_{hist}.currentYear$, which is only established because another step T_2 empties *events* whenever *year* is incremented, thereby establishing an implicit relation between entries in *events* and the current value of *currentYear*. If this stronger property is required for the proof, then the user must provide a provenance witness to inform Basilisk of this stronger relationship between *events* and *currentYear*.

While it is certainly possible to design protocols that pathologically attack this weakness, we did not observe many implicit relationships in our real-world examples. In particular, the only implicit relationship that we observed in real protocols is when hosts reset certain collections (e.g., a set of votes) when they enter a new epoch, as is the case with all three protocols that required Provenance hints in Table 1.

Second, the strength of Provenance Invariants generated by the atomic sharding algorithm corresponds to the accuracy of the estimated footprints. Overestimation of footprints may lead to steps being spuriously associated to atomic shards. These steps are then added to the disjunction expression in the generated Provenance Invariants, weakening these invariants. The user may then manually strengthen the generated invariants using provenance witnesses. In our experience, however, we did not face this issue with any of the protocols to which we applied Basilisk.

General limitations. Basilisk is not guaranteed to be complete. There may be instances of protocols and safety properties for which Basilisk cannot generate an inductive invariant strong enough to complete the proof, even with the help of user-supplied provenance witnesses. We did not come across such cases. If they arise, however, the user may augment Basilisk's generated invariant with manually-written clauses.

In addition, the protocol's safety specification is trusted, i.e., assumed to be correct. Also, while the protocol description is verified, a developer must trust that the description accurately models their actual system, or employ prior techniques [10,36] to provably connect it to their implementation.

We also assume the correctness of Dafny and all its underlying dependencies and hardware. A successful proof is only as trustworthy as these components.

Finally, Basilisk is designed to prove the safety of crash fault tolerant distributed protocols. Liveness proofs and weaker fault models are beyond its scope.

7 Related Work

Many techniques to verify distributed systems have been proposed in recent years. Historically, these techniques either allow arbitrary protocol descriptions at the cost of developer effort to manually find and prove inductive invariants, or provide significant automation by forcing the developer to fit their protocol description into a decidable logic.

7.1 Using Undecidable Logic

Early frameworks for verifying protocols written in undecidable logics entirely relied on the developer to devise and prove inductive invariants [10, 36]. Similarly, recent frameworks like Aneris [16] and Grove [31] support verification of distributed systems using separation logic. They combine reasoning about implementations and protocols, but require the developer to write and prove all of the invariants by hand.

Some approaches partially automate the proof process, but they do so by restricting the allowed protocols. Examples include Pretend Synchrony [35], PSync [6], the $\mathbb{CL}$ logic [5] Model [4], and *ConsL* [22].

The Message Chains work [23] proposes a structure for invariants that focuses on messages, and they use specification mining over randomly generated executions to infer some of these invariants in their benchmarks. Our Provenance Invariants connect host state to protocol steps, and can be derived statically from the protocol description instead of using specification mining. Our static analysis approach works for all of our benchmarks. Message chains are potentially complementary if a protocol cannot be verified with Basilisk's automatically generated invariants and a message chain invariant would help.

The most recent and related work to Basilisk is Kondo [40]. Both Basilisk and Kondo help the developer identify invariants when verifying distributed protocols in a Dafny-based setting. Kondo's invariant taxonomy introduces two top-level classes of invariants: Protocol Invariants and Regular Invariants. Protocol Invariants are inter-host or intra-host properties that do not talk about the network. Crucially, Kondo requires the developer to manually derive and prove Protocol Invariants, which remains a creative and laborious task. In contrast, Basilisk automatically proves such properties via Regular Invariants (§3). We leave the theoretical exploration of the logical relationship between Protocol Invariants and Basilisk's generated Regular Invariants to future work.

Basilisk also inherits Kondo's automation for two subclasses of Regular Invariants, Monotonicity Invariants and Ownership Invariants. However, this paper proposes Provenance Invariants, which replace and generalize Message Invariants in Kondo's taxonomy. Network-Provenance Invariant correspond to Kondo Send Invariants. Host-Provenance Invariants subsume Kondo's Receive Invariants, and we also generalize them to properties whose provenance is a local tran-

sition rather than a message receive. Finally, Kondo does not fully automate deriving Receive Invariants, whereas Basilisk does via the atomic sharding algorithm (§4).

7.2 Using Decidable Logic

Many frameworks and algorithms rely on protocols and invariants expressed in EPR, a decidable fragment of first-order logic [8, 14, 15, 21, 27, 28, 33, 37, 38]. These all share a critical limitation, in that they require the developer to write their protocol within a restrictive language, which disallows quantification of the form $\forall\exists$, as well as theories like arithmetic. Translating protocols into EPR requires expertise and is not always possible [26].

8 Conclusion

This paper presents a general class of Provenance Invariants that can be systematically derived from a distributed protocol and can be used to prove interesting inter-host properties by, for example, chaining a Host-Provenance Invariant that shows a message was received, with a Network-Provenance Invariant that connects the message to the sender's state. We develop an atomic sharding algorithm to find Provenance Invariants given only a protocol description. We implemented atomic sharding in a tool called Basilisk, which is able to generate inductive invariants sufficient to prove the safety of 16 distributed protocols without any user-defined invariants.

Acknowledgment

We thank our shepherd, Sudarsun Kannan, and the anonymous OSDI reviewers for their great feedback. This work was supported by the National Science Foundation grants CCF-2318953 and CCF-2318954.

References

[1] Dafny version 4.2. Available online at: https://github.com/dafny-lang/dafny/releases/tag/v4.2.0.

[2] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2021, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[4] Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22, Apr 2009.

[5] Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)*, 2014.

[6] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A partially synchronous language for fault-tolerant distributed algorithms. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[7] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 405–425, Cham, 2019. Springer International Publishing.

[8] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, page 131–150, Berlin, Heidelberg, 2021. Springer-Verlag.

[9] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, Apr 2021.

[10] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2015, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, Jun 2017.

[12] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[13] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *J. ACM*, 64(1), Mar 2017.

[14] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 338–356, Cham, 2022. Springer International Publishing.

[16] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In Peter Müller, editor, *Programming Languages and Systems*, pages 336–365, Cham, 2020. Springer International Publishing.

[17] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[18] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[19] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, Dec 2001.

[20] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2019, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)*, 2017.

[23] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2), Oct 2023.

[24] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 2015.

[25] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, Jun 2014. USENIX Association.

[26] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[27] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. Induction duality: Primal-dual search for invariants. *Proc. ACM Program. Lang.*, 6(POPL), Jan 2022.

[29] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, Apr 2010.

[30] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL), Dec 2017.

[31] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2023, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.

[32] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng.*, 9(3):219–228, May 1983.

[33] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, Santa Clara, CA, Jul 2024. USENIX Association.

[34] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 662–677, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2019.

[36] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 485–501, Carlsbad, CA, Jul 2022. USENIX Association.

[38] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, Jul 2021.

[39] Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. The Kondo tool. Available online at: `https://github.com/GLaDOS-Michigan/Kondo`.

[40] Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. Inductive invariants that spark joy: Using invariant taxonomies to streamline distributed protocol proofs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 837–853, Santa Clara, CA, Jul 2024. USENIX Association.

A Artifact Appendix

Abstract

The artifact represents the source code of the Basilisk prototype. It also contains the descriptions of the protocols used in the evaluation (§6) of Basilisk, together with the scripts and proof annotations required to verify each protocol.

Scope

The artifact lets readers validate how Basilisk relieves developer effort in verifying distributed systems. This is measured by the following criteria:

1. The user only needs to provide a few simple hints for Basilisk to automatically generate an inductive invariant for each protocol.

2. The user should be responsible for writing fewer lines of proof code, compared to Basilisk's predecessor Kondo.

Contents

This artifact has two main directories.

local-dafny. The directory local-dafny/ contains the source code and executable for the Basilisk tool. It is developed on top of the Kondo artifact [39], which itself is a fork of Dafny 4.2.0 [1]. The core Basilisk functionality is implemented in the local-dafny/Source/DafnyCore/Basilisk/ sub-directory.

basilisk. The directory basilisk/ contains the descriptions of protocols on which Basilisk is evaluated (§6). Each protocol is also associated with an example user-written safety proof (step ❹ of Figure 5) and a script to run the verification commands.

Hosting

The artifact is hosted on GitHub: https://github.com/GLaDOS-Michigan/Basilisk (commit version fd0bc04). A copy of the repository is also hosted on Zenodo: https://zenodo.org/records/15392830.

Requirements

The artifact was developed and tested on an M3 MacBook Pro running MacOS Sequoia.

List of Protocols Evaluated

In this section, we describe each protocol listed in Table 1, and their associated safety property.

Echo Server. A server host receives messages from clients and responds to each message with an identical message.

Safety property: The set of responses a client has received from the server is always a subset of the messages it has sent.

Ring Leader Election. The classic Ring Leader Election protocol [3].

Safety property: At most one host is elected as the leader.

Simple Leader Election. Hosts request votes from peers and votes for a single peer [34]. A host declares themselves leader if they receive a majority of votes.

Safety property: At most one host is elected as the leader.

Paxos. The classic Paxos protocol [19].

Safety property: No two hosts can learn different values.

Paxos-Combined. Variation of Paxos where each host process assumes all three of leader, acceptor, and learner roles.

Safety property: No two hosts can learn different values.

Paxos-Dynamic. Variation of Paxos-Combined where leaders can dynamically increase their ballot number to retry their proposal.

Safety property: No two hosts can learn different values.

Flexible Paxos. The standard Flexible Paxos protocol [12].

Safety property: No two hosts can learn different values.

Distributed Lock. Hosts pass around a single lock in a ring configuration [10].

Safety property: At most one host holds the lock at any instant.

ShardedKV. A sharded key-value store where a shard may transfer a key to another shard.

Safety property: Every key is owned by at most one host.

ShardedKV-Batched. Same as the ShardedKV, except keys may be transferred in arbitrary-sized batches.

Safety property: Every key is owned by at most one host.

Lock Server. A single server host may grant a lock to one of many client hosts, and clients may return a lock it holds back to the server [21].

Safety property: At most one client holds the lock at any instant.

Two-Phase Commit. The classic Two-Phase Commit protocol.

Safety property: The conjunction of

1. All processes that reach a decision reach the same decision as the coordinator,
2. The Commit decision can only be reached if all processes prefer Yes, and
3. if all processes prefer Yes, then the decision must be Commit.

Three-Phase Commit. The classic Three-Phase Commit protocol [32].

Safety property: Same as Two-Phase Commit.

Reduce. A distributed protocol for a sum computation. Each host starts with a partial array segment, then computes the sum of the elements in their respective segment before sending it to a distinguished host. The distinguished host adds up all the values it receives from other hosts and returns the value.

Safety property: The value returned by the distinguished host is the sum of elements in the array formed by the concatenating all the hosts' array segments.

Raft Leader Election. The leader election phase of the Raft protocol [25].

Safety property: Each term has at most one leader.

Multi-Paxos. The classic Multi-Paxos protocol [19] where hosts reach consensus for a log of values rather than a singular value. Like Paxos-Dynamic, each host assumes leader, acceptor, and learner roles, and may increase its ballot number to retry proposals.

Safety property: For all entries in the log, no two hosts can learn different values.





Deriving Semantic Checkers from Tests to Detect Silent Failures in Production Distributed Systems

Chang Lou[1] Dimas Shidqi Parikesit[1,2] Yujin Huang[3] Zhewen Yang[4] Senapati Diwangkara[4]
Yuzhuo Jing[5] Achmad Imam Kistijantoro[2] Ding Yuan[6] Suman Nath[7] Peng Huang[5]

[1]*University of Virginia* [2]*Bandung Institute of Technology* [3]*Pennsylvania State University*
[4]*Johns Hopkins University* [5]*University of Michigan* [6]*University of Toronto* [7]*Microsoft Research*

Abstract

Production distributed systems provide rich features, but various defects can cause a system to silently violate its semantics without explicit errors. Such failures cause serious consequences. Yet, they are extremely challenging to detect, as it requires deep domain knowledge and substantial manual efforts to write good checkers.

In this paper, we explore a novel approach that directly derives semantic checkers from system *test code*. We first present a large-scale study on existing system test cases. Guided by the study findings, we develop T2C, a framework that uses static and dynamic analysis to transform and generalize a test into a runtime checker. We apply T2C on four large, popular distributed systems and successfully derive tens to hundreds of checkers. These checkers detect 15 out of 20 *real-world* silent failures we reproduce and incur small runtime overhead.

1 Introduction

Distributed systems today provide rich features through hundreds to thousands of APIs, parameters, commands, *etc.* However, various faults can cause a system to *silently* violate its semantics without explicit errors. For instance, a distributed message service that provides a publish API promising at-most-once semantics may deliver some message twice.

Despite the silent symptoms, such failures lead to severe consequences including data loss, wrong results, inconsistency, and vulnerabilities. The failure impact is further propagated to applications that rely on the affected semantics. The lack of explicit error signals allows these failures escape existing failure detectors [57, 67, 76]. Moreover, with few error messages, developers are clueless in debugging failures.

Ideally, bugs causing silent failures should be eliminated in testing. However, it is elusive to eliminate all bugs in production distributed systems. Indeed, a recent study shows that silent failures are prevalent in mature, extensively-tested distributed systems [68]. They take up *39% of all studied failures*. Recent studies from multiple cloud companies suggest that silent violations can come from not only software bugs but also mercurial CPU cores [37, 55, 84].

Therefore, there is a pressing need for *runtime verification* [53], a safeguard mechanism to continuously monitor a deployed distributed system and quickly detect silent failures at the scene. Central to this mechanism are a set of checkers that validate system semantics, *e.g.*, at-most-once delivery. We call them *semantic checkers* and distinguish them from generic checks (*e.g.*, for CPU usage, timeouts, and exceptions).

Since semantic checkers require system-specific knowledge, the conventional wisdom is that they have to be written *manually*. For large distributed systems, this is a daunting task. Unlike prior runtime checking works [32, 49, 65, 66] that focus on checking small programs or a handful of well-defined properties for protocols, the systems we target have abundant *loosely-defined* semantics. Even for semantics described in simple text informally, mapping them to a concrete, correct checker is not easy. One needs to deeply understand both the system code and behavior to decide what needs to be checked, what is expected, when should the check be triggered, *etc.*

The thesis of this paper is that it is feasible to automatically construct semantic checkers that are expressive and accurate for large distributed systems without significant manual efforts.

We observe that a distributed system's test cases often contain valuable information about the system semantics, which we can leverage to synthesize semantic checkers. There are solutions [41, 47, 52] that use statistical methods to automatically infer likely invariants from software execution traces (often obtained from running a test suite). Those inferred invariants only express program variable relations, which are too low-level to check system semantics. Many of the inferred invariants are false due to inaccuracies in statistical inference. A recent work [68] leverages regression tests for past failures to infer event relationship, but it has the same limitations.

Our key insight is that the *test code* encodes rich information, including semantics-specific oracles—*assertions*, and carefully-designed workloads (*e.g.*, Figure 1). This rich information is not leveraged in existing solutions. But if existing tests contain expressive checks, why do they still fail to prevent silent failures? This is because they are written in a way that is tied with specific input in a fixed setup, while the relevant semantics can be violated in production due to different bugs triggered by other inputs and dynamic conditions.

Based on the above observation and insight, we propose a novel approach that leverages the existing test codes developers already wrote and *directly transforms* them into semantic checkers. Our basic idea is to reuse the code skeleton of a

test case, but we relax its strict constraints to generalize the test into a *parameterized checker function*. The generated checkers can then be deployed with the system to production and activate under workloads different from the original tests to detect *new semantic violations*.

We design an end-to-end framework, T2C, that realizes this idea. T2C reduces manual efforts and generates checkers that are expressive and accurate to cover various system semantics.

Achieving the above objectives is extremely challenging. First, test code contains a mixture of instructions and many instructions are not applicable to a semantic checker. Second, test code is inherently written to check specific instances, while a useful checker needs to apply in other instances. Third, a test only applies in certain scenarios. We need to determine the precondition for the derived checker, which is not clearly specified in the test. The practices of writing tests also vary, even for tests from the same system.

It is important to recognize that, without domain knowledge, automatically generalizing all test cases into checkers is impossible. T2C aims to synthesize a significant number of checkers from existing tests to provide comprehensive coverage of semantics.

To understand the feasibility of our approach, we conducted a large-scale study on 210 test cases from six distributed systems. The study found that interestingly many test codes in practice have clear patterns to leverage for transformation and generalization, *e.g.*, the use of test utility function, simple relations between assertions and the workloads.

Guided by our study, T2C leverages a combination of static and dynamic analyses to transform existing test code into reusable semantic checkers. At a high level, T2C statically identifies the key checking logic embedded in test cases—typically assertions tied to important system behaviors—and repackages them into standalone *checker functions*. To ensure these checkers are meaningful beyond the original test context, T2C then dynamically analyzes the system and test code to trace relevant data and control flows. This enables T2C to extract the necessary *preconditions* under which the behavior should hold. By combining preconditions with checker functions, T2C produces semantic checkers that are both precise and generalizable. Finally, T2C performs multi-level validation on the synthesized checkers to ensure they are valid.

We have applied T2C on four widely-used distributed systems, ZooKeeper, Cassandra, HDFS and HBase. T2C successfully transforms 672 test cases in these systems into verified semantic checkers. We sample and reproduce 20 user-reported production silent failures, and the synthesized checkers detect 15 of them in a median of 0.188 seconds. Interestingly, most test cases from which the checkers are derived were added by developers long before the failures.

The main contributions of this work are as follows:

- We proposed a novel approach to synthesize semantic checkers from test code and presented a large-scale study on existing tests to validate the feasibility of this approach.

```
1  public void testSessionTimeout() {
2    DisconnectableZooKeeper zk = createClient(TIMEOUT);
3    zk.create("/stest", new byte[0], OPEN_ACL, EPHEMERAL);
4    zk.close();
5    zk.disconnect();
6    zk = createClient(TIMEOUT);
7    Assert.assertTrue(zk.exists("/stest",false) != null);
8    Thread.sleep(TIMEOUT*2);
9    Assert.assertTrue(zk.exists("/stest",false) == null);
10   zk.close();
11 }
```

Figure 1: A test from ZooKeeper that validates the semantics of ephemeral znodes. The test creates an ephemeral znode (/stest), disconnects the client, and checks whether the znode is correctly removed after the session timeout.

- We designed T2C, an end-to-end framework that uses a hybrid of static and dynamic analysis techniques to realize the proposed approach for large distributed systems.
- We evaluated T2C on widely-used distributed systems, and showed its effectiveness on synthesizing expressive checkers that can detect production silent failures.

T2C is open-sourced at https://github.com/OrderLab/T2C.

2 Background and Motivation

In this section, we first use a real example to elaborate on our insight and proposed approach. We then present a feasibility study on existing test cases in popular distributed systems.

2.1 Example

ZooKeeper is a distributed coordination service that exposes hierarchical namespace of data called znodes. One feature it provides is ephemeral znode, the semantics of which guarantees that (i) an ephemeral znode exists for as long as the creating client's session, and (ii) it is deleted once the associated session ends. In a real-world silent failure [22], users observed that some ephemeral znodes were not removed even after the client session was long gone. This failure occurred *not* because the ephemeral znode semantics was not tested. Like other popular distributed systems, ZooKeeper has test cases for all major features including ephemeral znode. Figure 1 shows a test that exists prior to this failure.

However, the test cases are written to check only a few specific examples, *e.g.*, `/stest` in Figure 1, in a controlled setup. Passing the test cases does not imply that the system can obey the tested semantics once deployed. The production system in this case encountered a rare concurrency bug. This buggy condition does not appear in the test setup.

Nevertheless, the test does explicitly check the semantics. It includes two assertions (line 7 and 9) that describe the two guarantees of ephemeral znode. It also includes workloads that exercise the semantics (line 3–5).

To leverage this test for detecting failures in production, we may attempt to directly reuse it (with some changes so the `zk` client checks against the production instance instead of the testing instance). This would work if the ephemeral znode that violated the semantics happened to be exactly `/stest`, or if

Software	Language	Category	Version	Tests	
				Total	Sampled
ZooKeeper	Java	Coordination	3.4.11	434	35
Cassandra	Java	Database	3.11.5	4118	35
HDFS	Java	File Sys.	3.2.2	3265	35
MongoDB	C++	Database	7.1.0	1276	35
CephFS	C++	File Sys.	1.11.0	506	35
Mesos	C++	Cluster Mgr.	18.2.0	233	35

Table 1: Sampled test datasets from six studied systems.

the bug affected all ephemeral znodes. Neither was the case: only some znodes were affected in the production failure and they did not have the tested example path `/stest`.

The problem here is that the checking logic in the test is too restrictive. If we relax it, it is possible to retrofit the test into a more general checker. For example, by symbolizing the concrete arguments in it (`/stest`, *etc.*), we can transform the assertions and related statements into a parameterized checker function, which can be invoked with other arguments.

2.2 Feasibility Study

The example above is promising. However, not all tests are like it. Some tests may not have assertions for system semantics, or their assertions may be too specific to convert into a more general checker. To understand the feasibility of our approach, we conduct a study on existing test cases in six popular distributed systems shown in Table 1. We randomly sample 210 test cases to analyze, with 35 cases for each system.

For each case, we seek to answer the following main questions: (1) do tests contain useful semantic checks? (2) how is the checking done? (3) can the checking logic be generalized from tests to production settings?

The majority (87%) of the studied tests use standard assertions, such as `assertTrue` and `assertEquals`, as the checking mechanisms. The remaining tests do not have any explicit assertions. They are either performance benchmarks or checking by throwing an exception. These tests are typically not designed to expose semantic violations.

Finding 1: *Most sampled tests contain explicit assertions to check the system semantics.*

Hereinafter, we focus on the 183 tests that use assertions (which percentages presented will be based on). We summarize their checking approaches into several patterns (Figure 2):

- Check if the result of one operation matches the input to preceding operations, *e.g.*, the test writes something to a file, and checks by reading from the same file and comparing if the result matches the argument in the write operation.
- Check the consistency of a workload. For example, one HDFS test case selects several target nodes for storing a file; it then uses an assertion to verify whether these target nodes reside in different racks.
- Check whether the return result matches the impact of some operation or expected status of the system. For example,

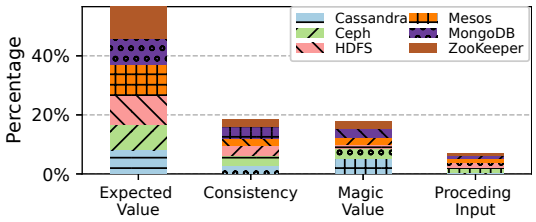


Figure 2: Assertion type distribution (Finding 1).

`testAbortNewFileAfterFlush` writes a file but shortly aborts it; it then verifies that the destination directory is empty: `assertEquals(0, testDir.list().length)`

- Comparing the result of an operation with a "magic value", which is a constant predefined by developers. For example, `ASSERT_EQ(url_unescape("foo%20bar"), "foo bar")` checks the `url_unescape` operation correctly decodes the input string. The constant `"foo bar"` is a magic value.

In the first three patterns, the checking is based on contexts within the tests. For the last pattern, the magic value is hard-coded, so deducing why it is expected can be difficult. Among the 183 tests, 37 (20%) tests have the last pattern.

Finding 2: *Checking is typically based on contexts within a test, but 20% tests check by comparing with magic values.*

Since the assertion checking involves comparing with an expected value, we examine the constraints between the expected values in assertions and other values in the test to figure out how to get the expected value in a semantic checker.

Interestingly, in many tests, the expected values in assertions do *not* need to be changed even if the workload varies, such as `true`, `0`, or system configuration. Figure 1 is an example: even if the path changes, the expected value `null` holds.

When the expected values do vary, it is common for them to have simple *equality* relation with other values in the test. For example, a Cassandra test uses an assertion `assertEquals("CREATE ROLE role1",obfuscate("CREATE ROLE role1"))`. The expected value can be simply inferred from the argument to the `obfuscate`. In other cases, the assertions use a defined variable from the workload as the expected value. Thus, we can directly find the source of the expected value.

The constraints in 117 (64%) tests fall into the above three categories (constant, equality, same variable). In another 26 (14%) tests, the constraints are simple arithmetic relations, such as greater than, or smaller than.

Finding 3: *In the majority of the tests, the constraints of the expected values are simple, because they do not need to change, or equal some other values, or directly use variables defined in the workload, or are basic arithmetic relations.*

In the remaining 40 (22%) tests, the constraints cannot be inferred without domain knowledge. They use magic values that do not have simple relations with other values in the test. For example, in ZooKeeper test case `testValues`, to get the expected values in `assertEquals(14L, values.get("sum_key2_test")`, we need to guess the operation from the string, and then calculate

the result. These steps cannot be automated without further knowledge. The assertion for `url_unescape` is another example.

Finding 4: *In a non-negligible number of tests, the expected values cannot be easily inferred from other values in a test.*

Besides using assertions directly in the test function, 40 (22%) tests invoke some utility functions that perform the checking. These functions are written to ease setup or result checking and are modular. They contain rich domain-specific logic. One function is often used by multiple test cases.

Finding 5: *Using utility functions is a common practice in tests, which can be leveraged in creating semantic checkers.*

We explore if the checking logic can be generalized from tests to production by replacing specific instances. Some tests use magic values, making them hard to generalize. Even if expected values are replaceable, assertions may not apply in production due to special configurations. For example, in the Cassandra test `createSuccess`, the assertion `assertFalse(tmd.params.compression.isEnabled())` only works if compression is disabled.

Interestingly, we find the tests to be more general than we anticipated. Of the 183 tests with explicit assertions, 65% satisfy both requirements of generalization: (1) the constraints of expected values are easy to infer; (2) the checked semantics are not limited to just special configurations or corner cases.

Finding 6: *Almost two thirds of the tests have checking logic that can be generalized to production.*

Finally, we examine if the assertions in the tests involve disruptive actions, such as deleting files or restarting the cluster. If so, invoking the checkers in production may pose risks. While many test cases involve such dangerous operations, they are part of the workload. When we retrofit a test into a semantic checker, the checker does *not* exercise the workload. It waits for the clients or users to execute these operations in production, and activates the assertions afterward.

Finding 7: *Although a test workload may contain disruptive operations, the assertions do not perform such operations.*

2.3 Implications

While transforming tests into runtime checkers is an ambitious goal, our study shows that it is promising. Existing tests contain valuable semantic information. Since most tests use assertions as the checking mechanisms (Finding 1), we can recognize their checks from standard interfaces. Findings 3 and 6 imply that deriving a checker function more general than the test is feasible, as the expected values can adapt to different instances. Moreover, the values used in assertions often have simple relations with other values in the workload, thus inferring the constraints to avoid over generalization is possible. The modularity from utility functions commonly used in test cases (Finding 5) also eases the transformation.

Nevertheless, we identify two primary challenges with this approach. First, a test contains code that both exercises a system and checks if the system behaves properly, while a checker is only responsible for the latter. Thus, our solution needs to decouple the two and make the checks standalone. Second, identifying the checker precondition is not easy. The checker should only be invoked when the production system exercises the semantics relevant to the test. In addition, the original test only uses a few example inputs, but the checker should also apply on inputs from production.

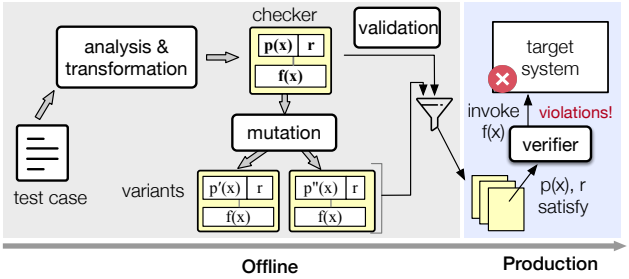


Figure 3: Workflow of T2C.

In the general case, both challenges are too hard to address automatically without domain knowledge. Some restrictive tests are also unnecessary or not applicable to run in production. We thus do not target transforming all tests but instead aim to transform a decent number of promising tests to fill the void of semantic checkers in large distributed systems.

3 Overview of T2C

Informed by our study, we design T2C, an end-to-end framework that transforms existing test cases into semantic checkers to detect silent failures in production distributed systems. Our framework assumes that checkers derived from specific test inputs or setups generalize reasonably well to broader scenarios encountered in production. This assumption is supported by our empirical study (Section 2.2).

Different from existing solutions, T2C directly leverages the test code and takes a *transformation* approach. It preserves the code structure of a test while extracting and generalizing the essential checking logic. T2C does not require a large number of traces for statistical inference. Moreover, unlike the state-of-the-art solution that relies on regression tests of past failures [68], T2C targets any unit tests or integration tests. T2C aims to not only reduce the manual effort but also generate checkers that are accurate and expressive enough to capture a wide variety of system semantics.

T2C is designed to be a monitoring system, where checkers only provide failure alerts and diagnosis clues to operators; they do not automatically mitigate failures (by crashing or restarting the system). This scope is the same as existing monitoring tools. T2C enhances existing solutions with stronger checks to reliably detect silent failures with low false alarms.

Problem Statement. Given a test case T, T2C aims to derive a runtime checker C that verifies the system semantics tested by T. The checker C consists of three components: (1) C_f, the

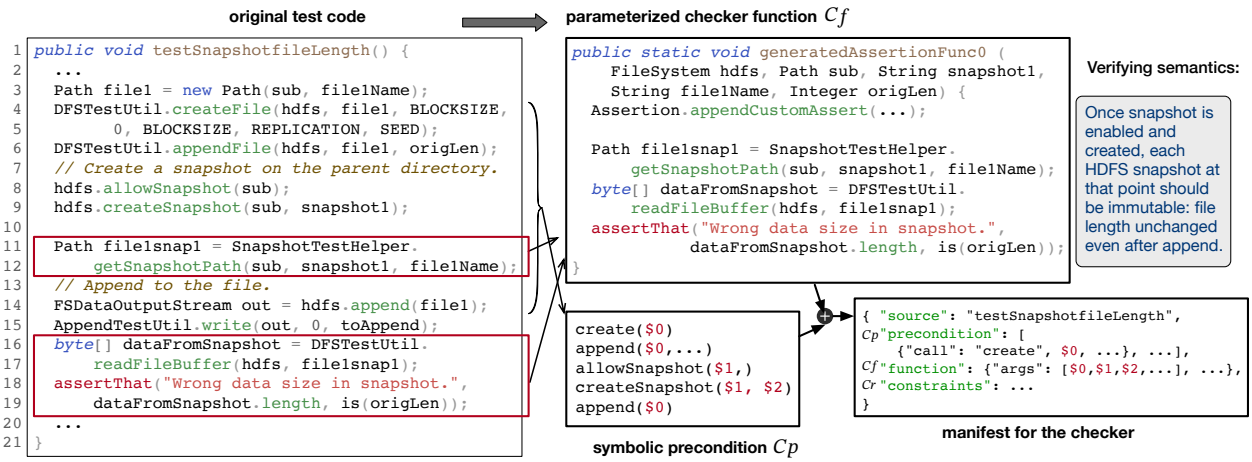


Figure 4: Example of checker T2C generated for a test in HDFS.

parameterized checker function; (2) C_p, the symbolic checker precondition; (3) C_r, the constraints that include relations between parameters, constant values for specific parameters, and system configurations.

Input and Output. To apply T2C on a new system, developers need to provide a configuration file that describes the code structure information (*e.g.*, path to system packages and test packages) and the compilation instructions. T2C will use these information to automatically analyze and execute tests. In addition, developers need to specify a few core system classes for instrumentation.

T2C outputs a list of checker functions as well as one manifest file (in JSON format) for each checker function. The manifest file records metadata for the checker, such as the source test name, the precondition, and the constraints.

Workflow. Figure 3 shows the high-level workflow of T2C, which consists of two phases: *offline* and *production*.

In the offline phase, T2C performs static analysis and dynamic analysis on the given test case T. Figure 4 shows an example of the semantic checker that T2C automatically derives from a non-trivial HDFS test case. It identifies the checking related instructions in the test, and transforms them into a standalone checker function C_f. It also derives the symbolic precondition C_p as well as the constraints C_r. During this process, a key goal is to relax the constraints in T, so that the checker can apply on more general scenarios. However, if T is written too restrictively, the derived C may not be general enough. To address this problem, T2C further introduces a mutation step that aims to produce variant checkers by applying some changes to C_p and C_f. After generating a set of candidate checkers from existing tests, T2C performs a validation step to filter invalid checkers. Developers can then further inspect and select from the validated checkers to deploy together with the target system.

In the production phase, the system runs with the selected checkers and the T2C verifier. The verifier monitors the system workload, and evaluates whether any checker's precondition C_p and constraints C_r satisfy. If so, it activates the checker function C_f and passes the concrete arguments accordingly. When some checker fails, the verifier generates an alert and a debugging report about the semantic violation.

4 Checker Generation and Validation

This section describes the T2C designs for its *offline stage* (Figure 3): checker generation and validation. Checker generation contains several steps as Figure 5 shows.

4.1 Encapsulate Checker Function

A test case contains many instructions such as preparing input and invoking system APIs. To construct a semantic checker, T2C is most interested in extracting the essence—the test oracle instructions, which contain the correctness criteria for certain system semantics. As our study shows, most test oracles are written as assertion instructions.

However, a test oracle instruction alone cannot be directly used as a runtime checker. The checking logic in the test code typically contains a series of instructions, so only including the oracle instruction would be incomplete and not even executable. For example, in Figure 4, the test oracle is in line 18 (`assertThat`), but it depends on several other instructions including calling a helper function to get the snapshot file path, and defining variable `dataFromSnapshot`.

T2C uses static analysis to encapsulate the test oracle along with its related instructions into a *standalone checker function*. T2C identifies test oracle in the form of assertions. If a test does not contain any assertions, T2C will not generate checkers.

Algorithm 1 shows the core algorithm for the checker function construction. For a given test T, T2C first constructs a control-flow graph (CFG). It then runs *backward* data-flow analysis on the CFG. The analysis' goal is to construct a parameterized checker function C_f, and derive three things: (1) the set of parameters, S_a, for C_f's definition, (2) the set of intermediate variables, S_i, to declare locally in C_f, and (3) the list of instructions in T, S_s, to copy to the C_f function body. Other concepts in the algorithm align with classic dataflow

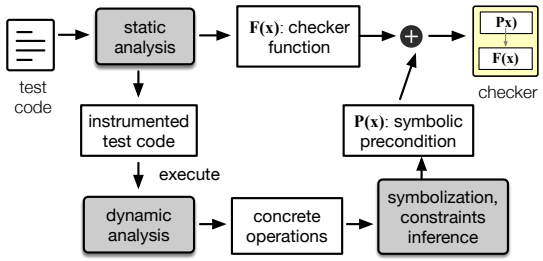


Figure 5: Checker generation workflow in T2C.

Algorithm 1: Checker function encapsulation.

Input: the target test `T`

Output: constructed checker functions, each C_f consists of: Set S_a of parameters in the generated function, Set S_i of intermediate variables, Set S_s of instructions to copy to the generated function

foreach $T_o \in findTestOracles(T)$ **do**
  $S_a \leftarrow \emptyset$; $S_i \leftarrow \emptyset$; $S_s \leftarrow \emptyset$; $W \leftarrow \{T_o\}$;
  do
    Take s from W; **if** $shouldExclude(s)$ **then**
      continue;
    $Out[s] \leftarrow \bigcup_{s' \in succ(s)} In[s']$;
    $In[s] \leftarrow Gen(s) \cup (Out[s] \setminus Kill(s))$;
    $S_i \leftarrow S_i \cup In[s]$; $S_s \leftarrow S_s \cup s$; $W \leftarrow W \cup pred(s)$;
    $S_a \leftarrow In[s]$;
  while $W \neq \emptyset$
  $S_i \leftarrow S_i \setminus S_a$; $C_f \leftarrow output(S_i, S_s, S_a)$

analysis: `shouldExclude` determines whether a statement should be omitted from the checker. It is used to filter out operations that are dangerous for runtime checking, such as system restarts. `In[s]` and `Out[s]` represent the input and output dependencies of a statement s, capturing the relevant variables when s is reached (`In[s]`) and when s is exited (`Out[s]`). `Kill(s)` is a helper function that returns old variables removed by statement s. `Gen(s)` is a helper function that returns new variables introduced by statement s.

The analysis starts by identifying the test oracle (assertion) T_o. To compute S_s, we use the program slicing algorithm [86] which includes the T_o and its dependent instructions. The main dependencies are calculated based on S_i.

To compute S_a and S_i, the analysis identifies all the values used in each expression in T_o. Those values are to be defined in C_f. We need to distinguish whether they should be local variables or function arguments.

Our basic idea is that a value by default is in S_a (an argument), but if it is derived from other values, we move it to S_i and add its source values to S_a. To do so, the analysis recursively follows the use-def chains until it reaches values that are "out-of-scope"—either constants or variables defined outside the test case function. Those values are put in S_a, while the values along the chains are in S_i. The rationale for this analysis is that arguments naturally come from external sources, while local variables are to assist checker execution.

Take Figure 4 as an example. The algorithm will start from the assertion statement in line 18, which is simple to locate. This assertion includes two workloads with variables `dataFromSnapshot` and `origLen`. The algorithm's goal is to identify how to construct these workloads for runtime checking. Thus, it traverses backwards, analyzing each preceding statement. For example, when analyzing line 16, the algorithm finds out `dataFromSnapshot` is generated by a function call `readFileBuffer`. Thus the algorithm marks this statement to include in the checker function, and add the parameters of `hdfs` and `file1snap1` to the tracking set instead of `dataFromSnapshot`. This process continues until the start of the test is reached. Eventually the algorithm marks all related statements (lines 11, 16, 18) to include, as well as a list of parameters (`hdfs`, `sub`, `snapshot1`, `file1Name`, `origLen`) for this checker function.

T2C also includes relevant branch instructions to S_s and variables used in the branch conditions in computing S_a and S_i. Thus, the derived C_f preserves T's control-flow structure.

At the end, T2C defines C_f with S_a and construct the checker body with S_s and S_i. T2C automatically fills in the missing parts including variable initialization and type conversion. It also handles name aliases—any statements that introduce aliases are included through data-flow propagation, since the algorithm constructs slices based directly on explicit data and control dependencies.

Side Effects. Our slicing algorithm does not provide soundness guarantee. It may include operations with side effects, *e.g.*, `write`. Some test instructions could be even dangerous to apply in production, such as `delete`. Those instructions are part of the original test workload but should not be included in C_f, because a checker should mainly observe production system state and not exercise workload. It should wait for some client or user to invoke such operations.

Our above algorithm is able to naturally exclude many of these instructions in computing backward slices for the assertions. The HDFS test in Figure 4 is an example. Although it includes instructions for creating snapshots, appending files, *etc.*, the checker function T2C derives only reads the file.

However, some instructions may still be included. For example, if an assertion directly checks the return value of a `write` call, the slice will include the `write` instruction based on the dependency. To avoid this situation, we refine the slicing algorithm to stop if it reaches a method that may introduce side effects to the system. We use purity analysis [80, 88] to construct a list of side-effecting operations. At the start of the static analysis phase, T2C analyzes the system call graph and outputs the side-effecting operation list, which is used in the encapsulation phase. We also add additional operations for each system. For example, we add three operations for HBase: "compact", "truncate" and "clean". Even if the list mistakenly misses some side-effecting operations, these operations will be exposed in the validation phase (Section 4.4).

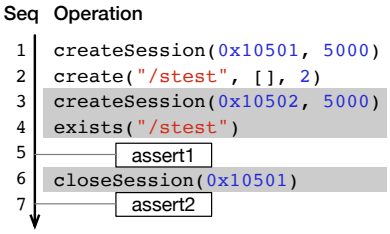


Figure 6: Concrete system operations from running test in Figure 1.

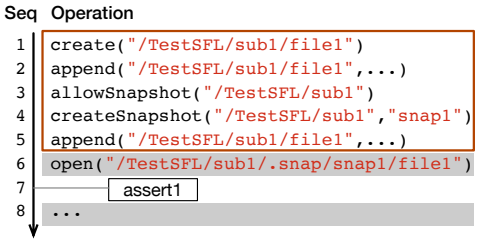


Figure 7: Concrete system operations from running test in Figure 4.

Multiple Assertions. One test may contain multiple assertions. T2C tries to derive one checker for each assertion separately. Although it can be appealing to combine multiple assertions in one checker function for efficiency reasons, it can introduce complexities. For example, if we encapsulate the two assertions in Figure 1 into one checker function, we need to decide whether to include the `Thread.sleep` instruction (line 8) in between them. By the slicing algorithm, we will not. But the second assertion will produce false alarms when the client session is not gone yet. Including this instruction is not perfect either, because the sleep timeout does not guarantee that the session is gone. The true operation to wait for before invoking the second assertion should be the system operation `closeSession` (seq. 6 in Figure 6), which is initiated by the system rather than the checker. By separating the two assertions into two checkers, we avoid the complexities. The second checker uses `closeSession` as its precondition.

Differences with Traditional Program Slicing. Our algorithm has a few key differences from traditional program slicing [86]. Program slicing includes all instructions that influence an assertion, even if they modify the system. T2C refines slicing by stopping at side-effecting operations using a combination of static purity analysis and a manually curated list of known unsafe operations, ensuring that the checker remains observational. Program slicing may include all control-flow branches affecting an assertion. Besides extracting relevant instructions S_s, T2C also identifies (1) the set of parameters, S_a and (2) the set of intermediate variables, S_i for checker construction. Program slicing merges all assertion-dependent statements into a single slice. T2C separates each assertion into its own checker to avoid unintended interactions.

4.2 Identify Checker Precondition

The checker function generated in the above step will be standalone. However, it should only be invoked when the production system exercises the semantics underlying the original test. For example, in Figure 4, it will check the HDFS snapshot's file immutability semantics, but it is only applicable if some client in production has requested HDFS to take a snapshot using relevant APIs. Essentially, each C_f has a *precondition* C_p (our usage of the term slightly differs from Hoare logic), which T2C will derive.

Approach. Our basic idea is to extract code before the assertion that does not directly impact assertion values, treating it as preconditions. However, setup code like cluster startup is irrelevant to runtime checking and should be excluded. Therefore, T2C focuses on relevant operations after system runs, targeting *system APIs* used during tests. Since static analysis struggles with identifying target APIs and their arguments due to wrappers and polymorphism, T2C uses *dynamic analysis* on test code to derive preconditions.

4.2.1 Obtain Concrete Precondition

T2C includes an instrumentation library. It adds hooks to the system interfaces to record information including the operation type as well as the concrete arguments used. We focus on entry interfaces on the *server code*. For example, in HDFS, T2C instruments public methods in the `NameNodeRpcServer` class, such as *create* and *createSnapshot*. T2C then executes the test code and records the instrumented information.

After the test execution, T2C obtains an ordered sequence of system operation invocations, each with concrete arguments. The next question is *how to determine the concrete precondition from this sequence?* T2C treats the subsequence that occurs before the execution of T_o as the candidate precondition. To identify this subsequence, the instrumentation library adds hooks to T_o in the test code as well, which emits a record of T_o's execution in the sequence. However, this subsequence likely also contains operations that are exercised to provide arguments to T_o. These operations should be *excluded* from the precondition, because they are initiated by the checking. In other words, they belong to the checker function C_f constructed in Section 4.1.

Take Figure 6 as an example, which is the sequence of operations T2C obtains when running the test in Figure 1. The 5th operation is the marker T2C emits for the assertion in line 7 of Figure 1, so that candidate precondition is the first four operations. However, the 3rd and 4th operations provide arguments to the assertion, so T2C excludes them. Similarly, for the sequence T2C obtains from the test in Figure 4, the precondition includes only the first five operations (Figure 7).

Figures 6 and 7 also show that the operations T2C obtains in the dynamic analysis are at the system side, which can have differences with the static names appearing in the test code, because the test uses many wrappers. Focusing on system-side operations provides a clean way for T2C to deduce the precondition despite many test-specific methods. It also allows the precondition to apply in the production system.

While our heuristic excludes computations that do not directly affect assertion parameters, these computations may

still affect indirectly, *e.g.,* sorting an array. Such operation will be either captured during dynamic analysis as precondition (if sorting is instrumented) or during encapsulation as part of the checker (if sorting is a library call used in the assertion).

4.2.2 Symbolize Concrete Precondition

The precondition determined from Section 4.2.1 is tied with the specific workloads in the test. Such precondition is not very helpful for the in-production checker, because it cannot apply to scenarios beyond the original test. For example, the concrete precondition T2C extracts for Figure 1 (also Figure 4) is tied with one fixed path and does not apply to other paths.

To address this problem, T2C uses a novel idea to symbolize the concrete values and turn C_p into a symbolic precondition. However, we cannot simply turn each concrete value into a separate symbolic value. We need to identify the constraints among them, which is challenging in the general case.

Fortunately, our study (Section 2.2) shows that many values in tests simply have an *equality* constraint, which is easy to infer. When encountering a concrete value in a clause that is equal to an earlier value, T2C assigns the same symbol rather than a new symbol. Besides *equality*, T2C also supports simple constraints such as *less than* and *contains*, which our study shows is also common. For such constraints, T2C will add them to the checker manifest file, which will be evaluated as part of the precondition at runtime.

Our constraint inference is not perfect. Two concrete values that are equal may be a coincidental choice by developers, which can over-constrain the C_p during symbolization. For example, we may symbolize `foo("/stest","/stest")` into `foo($1,$1)`, but the second argument can be arbitrary. It is possible to leverage more information in the test code to improve the accuracy and support more complex constraints. We leave this improvement as future work.

Our tool by default does not symbolize enum type parameters as workloads, based on the observation that enum type parameters are usually part of interface semantics.

4.2.3 Mutate Precondition

The above symbolization step generalizes the precondition. However, the symbolic precondition can be unnecessarily restrictive. This happens because the original test only uses one specific sequence of operations, while the tested semantics still applies when the sequence changes (slightly). Thus, the production system can violate the semantics under a similar triggering condition. For example, instead of calling `append` *once* after `createSnapshot` as the test in Figure 4 does, a client in production invokes `append` multiple times. In this case, the immutability semantics still applies. Concurrency may also affect the checker generation process. If the test execution is nondeterministic, the generated checker results may have different preconditions corresponding to different execution sequences. If we use the symbolic precondition exactly the same as the test, the derived checker will not be triggered when it can detect the violation.

To address this problem, T2C further relaxes the constraints of the precondition by *mutating* it. T2C currently supports four types of mutation: *reduce* (remove an operation), *insert* (add a wildcard that matches with any operation), *duplicate*, and *reorder* (swap the order of two adjacent operations). For the above example, we can apply the *duplicate* mutation to the `append` operation in the concrete sequence. T2C takes a simple enumerative (bounded) approach to try different mutations in different positions of the precondition sequence. Each mutation produces a *variant* precondition $C_{p'}$. It then relies on the validation step (Section 4.4) to evaluate whether the variant preconditions are necessary and valid.

4.3 Additional Handling

Control Flow Assertion. We observe that control-flow assertions are also common in tests. Unlike other assertions, control-flow assertions are designed to be not executed, *e.g.*, an `Assert.fail`. Thus, we can not simply instrument the assertion itself. To support this type of assertion, T2C flags the method call before such an assertion. When that statement is executed, T2C emits a special marker in the operation sequence to indicate that the next operation should fail.

String Queries. Some systems may not have clear operation interfaces. For example, Cassandra takes input queries as strings and uses ANTLR to generate parsers. The processing logic is complex and tightly coupled with other internals, making it difficult to find instrumentation points. We added a simple parser that interprets string queries to different system operations and extracts the operations' parameters.

System Constraints. We observe that some tests are indented for only specific system configuration and do not apply when the system configuration is different. For example, write operations should not be allowed in "read-only" mode. Thus, we also need to extract such system configuration constraints. We find that most configurations are defined in the setup phase of tests, such as `System.setProperty("readonlymode.enabled", "true")`. Before the checker building phase, T2C will search in test class bytecodes with dynamic analysis libraries, identify such system property and record in the generated checkers.

4.4 Validate Checkers

Like manually written checkers, the checkers T2C generates may not always be correct. For example, the encapsulated checker function may miss some instructions due to the inaccuracies in static analysis; a complex constraint existing in the test may not be inferred, so the precondition is too loose.

T2C uses four levels of validations to prune invalid checkers. The first level is to ensure the derived checker function passes compilation, *i.e.*, syntactic correctness. The second level is to ensure the checker function is runnable by leveraging JVM verifier, which performs various sanity checks.

The third level is a more interesting "self-validation" approach. T2C first replaces the original assertion statements

in the test with the generated checker function. It then runs the modified test. The checker function is supposed to pass because a specific workload should apply in a generalized version. If the checker function cannot pass its own test, it is problematic, so T2C marks it as invalid.

Self-validation alone is not enough, so T2C includes a cross-validation step that simulates a pre-production environment to assess checker quality. The cross-validation step aims to remove "over-generalized" checkers, meaning they incorrectly generalize test-specific invariants to broader runtime contexts. Specifically, during cross-validation, we run each checker against workloads from other passing tests (different from the test from which the checker originated). If a checker incorrectly triggers alarms during these unrelated but correct workloads, it indicates that the checker is overly general and thus producing false positives. Such problematic checkers are subsequently filtered before deployment. This cross-validation is particularly effective at filtering out checkers that over-generalize semantic-specific workloads (*e.g.,* creating a list of size 0 to check if it is empty, where 0 should remain a constant). Such checkers are triggered excessively and are invalidated during cross-validation.

We apply several performance optimizations for the validation phase. Running all tests with all derived checkers can be time-consuming. T2C runs tests that cover most system functionalities first, so a bad checker can be invalidated quickly. It discards a checker that is already invalidated by some test to avoid repeated invalidation. We also implement a parallel mode that distributes the tests to run on multiple nodes, which significantly speeds up the checker generation and validation.

5 Deployment of Checkers and Verifier

T2C deploys checkers that pass the validation together with the target system to production. Developers can also inspect and select a subset of validated checkers for deployment. T2C provides a verifier to manage the checkers in production. Its key components include an evaluator thread, a trace buffer and a precondition index, which are optimized with circular buffer and trie. For each checker, the evaluator uses the checker's manifest file to evaluate the symbolic precondition C_p and the constraints C_r against production workload. If both satisfy, the evaluator gets the concrete value for each symbol, and use them to invoke the parameterized checker function C_f.

The precondition evaluation requires information about the system state. This information is obtained by the instrumentation library described in Section 4.2.1. T2C links this library with the target system, which performs dynamic instrumentation to emit system trace in production. To evaluate the system constraints, the T2C runtime reads the related system configurations at startup time (*e.g.*, calling `System.getProperty("readonlymode.enabled")`) as part of the checker matching process. It then disables a checker if its required system constraint does not satisfy.

If C_f fails, T2C alerts operators about the possible failure

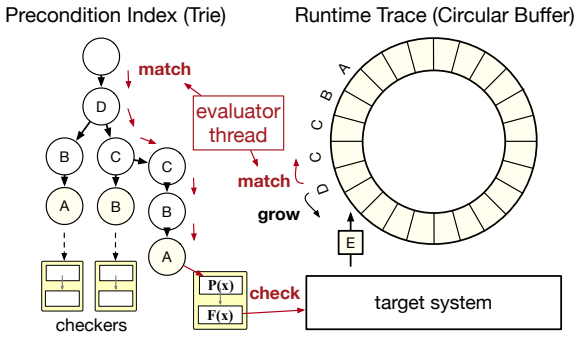


Figure 8: T2C runtime architecture.

along with debugging information such as the failed assertion and concrete arguments to help developers localize the bug.

T2C implements optimized runtime data structures (Figure 8) to reduce runtime storage and computation costs. The storage overhead mainly comes from bookkeeping system traces. T2C uses a circular buffer to manage maximum memory usage to avoid excessive GCs and automatically discard expired traces. The circular buffer design also avoids races between worker threads and the verifier. The computation overhead mainly comes from matching checker preconditions with runtime traces, which can be costly if implemented poorly (*e.g.,* loop through checkers) and generate backlogs. T2C implements a precondition index with a trie to accelerate matching. It reverses and inserts all checker preconditions (operation list) to the trie after loading checkers at the initialization stage. For example, to index three checkers, whose preconditions are `{opA, opB, opD}`, `{opB, opC, opD}`, `{opA, opB, opC, opC, opD}`, the tool constructs a trie as shown in Figure 8. The evaluator thread continuously matches patterns with the current trace suffix. This design effectively prunes unnecessary searches without traversing checkers one-by-one.

Dealing with Concurrency. Multiple clients may submit requests to the system simultaneously. As long as the preconditions satisfy, concurrency only affects the frequency of runtime checking since T2C checkers are activated based on the associated preconditions. For example, even if a checker is derived from a test that uses only a single client, if two clients issue requests to the same znode, they may also trigger the checker. Assume a checker's precondition is `s1, s2, s3`. This checker will still be triggered even if client1 sends requests that produce events `s1,s2` while `s3` is produced by client2's requests. Our evaluation benchmarks (Section 7) all use multiple clients and they do not prevent checkers from getting triggered. For concurrency in checker generation, T2C uses mutation to address this issue as described in Section 4.2.3.

Many system runtime properties, especially related to concurrency models, require special efforts to carefully design corresponding runtime checkers. T2C does not aim to check all system semantics—it automates the checker construction process by leveraging existing test assertions, significantly reducing developer effort. Although not every test assertion

can be safely transformed into a runtime checker, our approach has shown that many practical semantic checks can be safely extracted and deployed.

Checking Distributed Semantics. Some semantics involve operations across multiple nodes, which requires aggregating operations from different nodes. T2C follows two principles: assertions should run on their respective nodes; if an assertion involves workloads from multiple nodes, it should rely on RPCs, using return values from these calls to avoid introducing new issues. T2C provides a cluster mode to manage related requests across nodes in distributed environments. It aggregates traces from various nodes and, if a checker matches, sends messages to trigger assertions across nodes. This is achieved through an additional message layer, utilizing a distributed log library to forward event traces to the appropriate nodes.

Supporting System Setup Workloads. Some workloads used in the tests do not originate from normal client requests but rather from internal system setup. We refer to these as system setup workloads. We still need to capture such workloads for the checkers. For example, HDFS developers commonly use "MiniDFSCluster", a testing utility class that provides APIs for accessing internal system instances, state, configurations (*e.g.*, replication factor), or queries. To support these in production, we implement adapter classes that mimic the same interfaces but redirect the logic to interact with the actual production system. These adapters reside in the testing package and are invoked via reflection at runtime.

6 Implementation

We implement T2C mainly in Java with around 8,000 SLOC. The analysis and transformation component is implemented with Soot [83] program analysis framework. The dynamic instrumentation is implemented based on Javassist [34] bytecode editing library. The core designs of T2C, *e.g.,* the encapsulation algorithm, symbolizing arguments, are general and do not rely on Java-specific features. We believe porting T2C to support other languages, while requiring non-trivial engineering effort, will be relatively straightforward (*e.g.,* using LLVM for C/C++). T2C includes tests for its static analysis and runtime matching algorithms, as well as scripts to automate the library installation and test execution.

7 Evaluation

We conduct a comprehensive evaluation of our proposed approach. Our evaluation addresses several questions: (1) can T2C transform test cases in large distributed systems? (2) are the generated checkers useful in detecting silent failures? (3) do T2C checkers incur high false positives? (4) how fast is the T2C checker construction workflow? (5) what is the runtime overhead? The experiments are done in servers with 20-core 2.2 GHz CPUs, 64 GB memory, running Ubuntu 18.04.

7.1 Manual Effort of Using T2C

To use T2C, developers mainly need to provide information about the code structure (*e.g.*, paths to system and test packages) and compilation instructions. T2C uses this to automatically analyze and execute tests. Additionally, a few core system classes must be specified for instrumentation. This setup is minimal and a one-time effort. As a system adds new tests, T2C can generate more checkers without extra manual input. T2C also simplifies checker validation by automatically filtering out incorrect checkers, so developers only need to review noisy checkers flagged during pre-production testing.

Supporting a new system is straightforward. Our initial development of T2C uses ZooKeeper as the target system. Since then, T2C has evolved to be more general. Beyond the configuration file, developers supply a few side-effecting operations (around six per system) and testing utility adapters (about three classes per system). For example, integrating HBase took one person-week, mostly for testing and debugging.

7.2 T2C Checker Generation

To evaluate the effectiveness of T2C approach on real-world software, we apply the tool on four large systems: ZooKeeper (ZK), Cassandra (CS), HDFS (HF), and HBase (HB). We choose these systems because they are representative, widely used and have many user-reported silent failures [68]. These systems provide large test cases for T2C to analyze and generate checkers from. Each system is configured using a property file that contains 42 lines of codes on average.

T2C generates checkers for all four systems (Table 2). Each evaluated system has extensive tests (①) but not all tests are suitable for checker conversions. T2C targets a subset of the tests (②) that check system-level semantics and contain useful assertions. The excluded tests are either unit tests for data structures, or performance/stress tests tests with no assertions. We use package names to identify groups of the system tests instead of hand-picking individual tests. For example, most system tests in Cassandra are under `org.apache.cassandra.db` and `org.apache.cassandra.cql3`.

T2C succeeds in constructing checker functions for the majority of the target tests (③). It fails to convert about 7% target tests due to analysis errors in dealing with complex control flows for excluding side-effecting instructions. T2C runs the successful tests to obtain the preconditions. Upon finish generating the checkers, T2C performs sanity checks, *e.g.,* the checker structure must be complete, so only healthy checkers (④) remain. T2C then validates those checkers (Section 4.4), and retains only valid checkers (⑤). In the end, T2C generates 672 verified checkers in total. These checkers have 4.3 assertions on average. Note that this set *does not* include any tests that can reproduce failures in Table 5.

We manually inspect the generated checkers. They cover various system semantics, including request processing, storage, replication, compaction, *etc.* We observe that the completeness of the generated checkers depends on the coverage of the

System	Version	Classes	Tests			Checkers	
			①	②	③	④	⑤
ZK	3.4.11	123	428	109	100	90	46
CS	3.11.5	604	3283	257	242	232	100
HF	3.2.2	803	4304	816	729	707	230
HB	2.4	980	4287	990	948	904	296

Table 2: T2C checker generation overview.

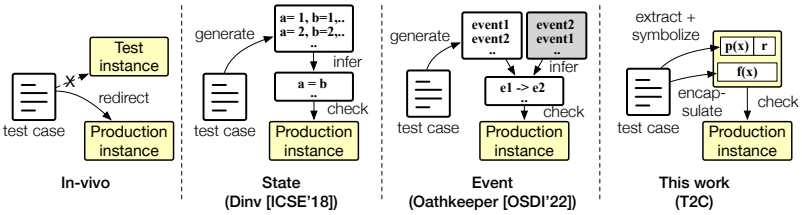


Figure 9: Comparison with the state-of-the-art semantic checkers (baseline).

system tests. With developers' continuous efforts of adding new tests, the checkers' completeness will further improve.

7.3 Detecting Silent Failures

T2C is designed to be a failure detection tool, which focuses on identifying abnormal symptoms at runtime. The key metric for a failure detection tool is once a bug is triggered in a running system and causing a failure, whether and how quickly can the tool detect the failure. This is different from bug-detection tools (*e.g.,* fuzzing), which focuses on creating the conditions, such as inputs and thread interleaving, to trigger a bug.

Failure Benchmark To evaluate generated checker effectiveness, we collect and reproduce 20 real-world silent failures from four systems. These failures are collected using the following methodology. We first query the evaluated systems' issue trackers with keywords "*silent*", "*corrupt*", "*inconsistent*", "*unexpected*", and "*incorrect*". This yields 6,190 issues (ZooKeeper: 410, Cassandra: 2054, HDFS: 1754, HBase: 1972). Then we randomly select 56 of them to inspect more closely. We read their descriptions to confirm whether the failures are indeed silent semantic violations. We exclude issues that are crashes, aborts, out-of-memory errors, or failures with explicit error signals (*e.g.,* an exception). This leads to a total of 24 cases. We attempt to reproduce each case and succeed for 20 cases (the four remaining cases lack the necessary information to reproduce). They become our evaluation failure dataset. Table 5 shows the list of evaluated cases. All of these failures lead to severe consequences. Each case takes one week on average to reproduce.

Baseline Detectors We evaluate each failure and compare T2C with three baselines (Figure 9): two state-of-the-art research solutions—Dinv [48] and Oathkeeper [68]—and one new semantic detector we develop.

- We design *in-vivo* checkers, a new checking method that continuously executes test cases (*e.g.,* create znodes) in the background. We implement such checkers by modifying the test case setup codes, and redirecting test workloads and assertions from simulated test instances to production server instances, so that tests can expose production issues.
- We implement *state* checkers based on Dinv [48], which infers key system state relations based on traces collected from test suite execution, *e.g.,* $lockSet_{t1} \cap lockSet_{t2} = \emptyset$. Dinv only supports systems written in Go language. We add support for Java programs.

		ZooKeeper	Cassandra	HDFS	HBase
In-vivo	Tests (total)	428	3283	4304	4287
	Tests (in-vivo)	391	56	1497	278
	Checkers (tests*)	94	46	73	78
State	States	131	470	1,244	691
	Data points	91,423	76,865	143,069	327,344
	Checkers (state invs.)	47,466	11,083	23,159	35,144
Event	Inferred failures	8	10	10	9
	Checkers (event invs.)	285	129	1209	1641

Table 3: Baseline checker metrics. *tests are sampled.

- We evaluate *event* checkers proposed by Oathkeeper [68], which infers semantic rules (*e.g.,* $lock \rightarrow unlock$) from past failures and uses the inferred rules to detect new failures. Oathkeeper supports ZooKeeper and HDFS. We port it to Cassandra and HBase.

We show the baseline checker metrics in Table 3. Similar to T2C, our methodology of converting in-vivo checkers targets tests that are applicable to intercept requests in client setup codes to redirect. This approach aligns better with test style in ZooKeeper and HDFS, which results in a higher applicable percentage in these two systems. We then sample a subset (291) of in-vivo checkers to deploy in production to shorten the time for checking a full round. For state checkers, their quantity is determined not by the number of test cases, but by the number of system states the tool monitors. More test cases yield additional traces of these states and enhance the inference accuracy. We generate 116,852 state checkers, most of which perform a simple arithmetic operation. For event checkers, their number is related to the number of past failures input into the tool. We generate a total of 3,264 event checkers.

Results Table 4 shows the detection results by different checkers. T2C detects most failures, in total 15 cases, compared to the baseline checkers. Tests we use to generate useful checkers were added long time ago by developers before the failures (3.9 years on average). In comparison, event checkers (based on Oathkeeper [68]) and state checkers (based on Dinv [48]) only detect five and three cases, respectively. Even the combination of all baseline checkers only detect eight cases. In general, T2C is particularly effective in detecting cases that require fine-grained assertions. While event and state checkers can expose some failures, they are limited to relatively simple relations. The in-vivo checkers are derived directly from unit tests without generalization—they reuse the exact concrete inputs and workloads originally used in

	ZK1	ZK2	ZK3	ZK4	ZK5	ZK6	ZK7	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5
T2C	0.112	1.018	0.460	✖	0.311	✖	✖	0.177	0.035	0.304	0.029	0.570	2.459	✖	0.039	1.715	✖	0.188	0.041	0.021
In-vivo	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	37.05	✖	✖	✖	✖	✖
State [47]	1.154	1.183	✖	✖	0.561	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
Event [68]	✖	✖	✖	✖	✖	3.433	3.569	✖	✖	✖	✖	✖	3.982	4.783	3.221	✖	✖	✖	✖	✖

Table 4: Detection results on 20 real-world semantic violations. *ZK*: ZooKeeper; *CS*: Cassandra; *HF*: HDFS; *HB*: HBase.

Id.	System	Sys. Feature	Description
ZK1 [17]	ZooKeeper	Ephemeral Node	Wrong Results
ZK2 [15]	ZooKeeper	Ephemeral Node	Wrong Results
ZK3 [16]	ZooKeeper	Read-only Mode	Illegal Operation
ZK4 [18]	ZooKeeper	Multi Request	Inconsistency
ZK5 [20]	ZooKeeper	Transaction	Inconsistency
ZK6 [19]	ZooKeeper	Deletion	Data Corruption
ZK7 [21]	ZooKeeper	Transaction	Data Loss
CS1 [4]	Cassandra	Range Query	Wrong Results
CS2 [3]	Cassandra	Static Row	Wrong Results
CS3 [1]	Cassandra	Time-to-live Data	Data Loss
CS4 [2]	Cassandra	Read Query	Wrong Results
HF1 [10]	HDFS	Snapshot	Broken Redundancy
HF2 [11]	HDFS	Erasure Coding	Data Loss
HF3 [12]	HDFS	Storage Management	Redundant Workloads
HF4 [13]	HDFS	Replication	Broken Redundancy
HB1 [5]	HBase	Memstore	Incorrect State
HB2 [7]	HBase	Table Update	Omission
HB3 [9]	HBase	Table Creation	Wrong Results
HB4 [8]	HBase	Time-to-live Data	Data Corruption
HB5 [6]	HBase	Reversed Scan	Wrong Results

Table 5: 20 real-world silent semantic failures reproduced for evaluation. *ZK*: ZooKeeper; *CS*: Cassandra; *HF*: HDFS; *HB*: HBase.

tests, rather than generalizing to variable runtime conditions as T2C does. As a result, in-vivo checkers only detect issues when the production workload exactly matches the specific scenario encoded in the original test.

T2C misses five failures. This is unsurprising since T2C cannot guarantee detecting all silent failures. It aims to extract semantics developers have implicitly encoded in the existing test code and generalize the checks for detecting a class of failures. For the missed failures, we find either the system lacks tests that cover the violated semantics, or there is some related test but contains no useful assertions or the test workloads are written in a spaghetti style (usually a result of several ad-hoc patches) that are hard to generalize. For such cases, if developers improve the test coverage and quality, the failures can still be likely detected. To demonstrate this, we add a test (Figure 10) for the missing detected case HB2. T2C successfully detects the case with the newly generated checker.

Case Studies **HF1** [10]: This is a running example in this paper. The constructed checker is shown in Figure 4. Developers observe that snapshots were modified after appending original files (Figure 11) due to a buggy implementation under certain configurations, which violates the immutable snapshot semantics. T2C detects the issue by comparing the returned snapshot size with the written size. State checkers cannot detect this issue since there is no corrupt state (the bug is in the read function implementation). Event checkers cannot detect the issue since normal and buggy traces are identical.

CS1 [4]: During an upgrade of Cassandra from 2.1.17 to 3.11.4, developers observe incomplete results from range queries due to a bug in the mixed mode implementation, which violates the semantics of range queries. T2C successfully detects and alerts on the incomplete result. State checkers cannot detect this failure since the bug lies in the query logic. Event checkers report no violations due to event traces.

ZK1 [23]: A transient network issue causes a delete proposal packet from the ZooKeeper leader to the follower to be dropped. A logic bug then prevents ephemeral nodes from being removed on the follower, even though the owner session has already expired. This violates the semantics that ephemeral nodes should exist as long as the session that creates the znode is active. T2C detects the ephemeral node still exists after the session expiration. State checkers detects this issue using ownership relations between ephemeral nodes and owner sessions. Event checkers cannot detect the issue since the deletion still happens, but the node is put back due to buggy synchronization.

ZK-new [24]: T2C is intended as a monitoring service for failures instead of bug finding, which is a complementary direction that focuses on crafting inputs and triggering conditions to test a system. Our evaluation thus focuses on evaluating T2C's ability to detect known, real-world failures from widely-used distributed systems. Nevertheless, during our evaluation, T2C successfully exposes a new bug in the latest ZooKeeper release (3.9.2). During our fault injection testing, T2C reports violations. With a close inspection, we observed that the ZooKeeper cluster was experiencing data inconsistency among different nodes (some ephemeral nodes never expire on certain followers). It is caused by `loadDatabase` loading a snapshot with a higher transaction id than the truncated log, causing the transaction replay to fail. T2C provided traces to help locate the root cause. We reported this bug to developers. This new bug is marked as "critical" (P1), which leads to data corruption and inconsistency, a severe consequence.

7.4 False Alarms and Side Effects

We evaluate the false alarm ratios of different checkers. To expose the corner cases that over time could occur in a real deployment, we run the system on Jepsen [14], a widely used testing framework for distributed systems. We implement clients in Clojure for all evaluated systems and define common APIs. Jepsen then generates random workloads and automatically injects network faults every 10 seconds. We cal-

```
1 public void testModifyTableSize() {
2   TableName tableName = TableName.valueOf("/test");
3   TableDescriptor desc = TestUtils.setupDesc();
4   admin.createTable(desc);
5   TableDescriptor newDesc =
    TableDescriptorBuilder.newBuilder(desc)
6       .setMaxFileSize(MAX_FILE_SIZE).build();
7   admin.modifyTable(newDesc);
8   assertEquals(MAX_FILE_SIZE,
    admin.getDescriptor(tableName).getMaxFileSize());
9 }
```

Figure 10: The test added for HB2.

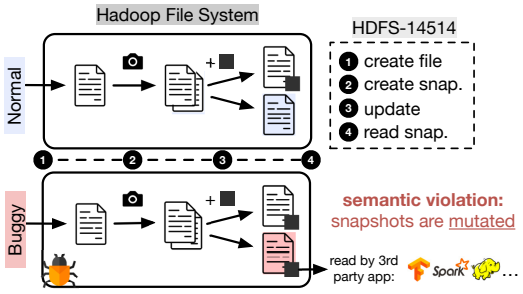


Figure 11: HDFS Snapshot Violation (HF1).

culate the false alarm rate as the ratio of total alarms reported in failure-free scenarios to the total times the checkers are triggered in a 30-minute duration.

The results are shown in Table 6. Overall, in-vivo checkers incur the highest false alarm rates (on average 47.4%) due to their inflexibility in adapting to differences between test and production environments. Meanwhile, with two filtering policies, static (exclude tests using specific APIs or configurations), and dynamic (exclude checkers report violations in dry runs), their false alarms can be reduced. State checkers also produce a high ratio of false alarms, which include false rules from test environments, such as `REQUEST_PREPROCESS_SAMPLE_RATE=0.1`. Some state rules are statistically probable but lack semantic soundness, *e.g.,* the value of `readRespCache.cacheSize` is not equal to `cnxnExpiryQueue.expirationInterval`, which involves two totally unrelated variables. Event checkers and T2C have a lower false alarm rate compared to the previous two types of checkers, thanks to the validation mechanism. The absolute numbers of failed T2C checks are: ZooKeeper (403), Cassandra (100), HDFS (24), and HBase (258). It is unnecessary for developers to analyze individual checking results, as most false alarms originate from a few noisy checkers. For ZooKeeper, nearly all false alarms come from two tests. Developers can address these by running a profiling test before deployment or adding a dynamic ban list.

On average, 56% of checkers generated by T2C are activated. The checkers T2C generates usually have more expressive and more complex preconditions, thus most of them will not be triggered at a high frequency. Thus, T2C does not need to incorporate probabilistic detection unlike other works such as Dinv. The absolute numbers of T2C checking are: ZooKeeper (31997), Cassandra (9905), HDFS (752), and HBase (41547).

	In-vivo Orig.	In-vivo +Sta.	In-vivo +Dyn.	State	Event	T2C
ZooKeeper	2.6	1.7	0	14.2	3.9	1.3
Cassandra	55.3	11.5	0.5	4.7	9.6	1.0
HDFS	68.9	35.4	0.1	6.2	9.3	3.2
HBase	62.6	44.0	1.9	22.8	17.9	0.6

Table 6: False alarms (%).

False negatives require a systematic approach to trigger silent semantic failures (which is a major research challenge) and an oracle to tell if a silent failure has happened. Our result in Table 4 provides some insights on what types of failures cannot be detected by T2C (*e.g.,* if the system does not have a test checking the violated semantics, or the test is of poor quality). Systems installed with T2C checkers do not experience noticeable anomalies, *e.g.*, file corruption, thanks to the filtering of unsafe APIs during checker construction.

7.5 Performance and Overhead

Offline Performance We measure the performance of each step in T2C 's workflow. Table 7 shows the results. The retrofitting phase usually finishes within minutes, which shows our static analysis algorithm is efficient even for large systems. The building and validation phase are more time-consuming as they require executing all test cases with intense instrumentation and logging.

Online Setup We next measure the runtime overhead of different detectors. We use popular benchmarks configured as follows: for ZooKeeper, we use an open-source benchmark with 15 clients sending 15 K requests (40% read); for Cassandra and HBase, we use YCSB with 40 clients sending 100 K requests (50% read); for HDFS, we use built-in benchmark NNBenchWithoutMR which creates and writes 100 files, each file has 160 blocks and each block is 1 MB.

Online Results As shown in Table 8, state checkers incur the highest overhead on system throughput, more than 50% on average. Its overhead does not come from the large number of invariants (most checker executions are arithmetic) but rather recordings for all tracked states. In-vivo checkers incur a 2.4% overhead, which comes from test workloads and checking in the background. Event checkers show a 1.8% overhead on throughput, since they only los event types and their the instrumentation is selective. In comparison, T2C generates a 4.0% overhead on average. The main contributing factor is low frequency of checker activation. As discussed in the previous experiment, T2C-generated checkers typically have precise and complex preconditions and the preconditions of many checkers are only infrequently satisfied at runtime. As a result, these checkers only run infrequently and this significantly reduces the overhead. Another key factor is from the efficient searches through our data structure optimizations.

We measure the CPU and memory usage under the baseline and T2C. The results show a moderate CPU usage increase: ZK (637.9% to 705.5%), CS (305.3% to 336.0%), HDFS (42.3%

	Retrofit	Build	Validate
ZK	25	4487	3837
CS	121	6774	6054
HF	78	23551	25170
HB	149	14401	13331

Table 7: Processing time in each step (s).

	In-vivo	State	Event	T2C
ZK	0.1	24.8	1.9	1.4
CS	3.0	59.0	1.8	9.1
HF	4.4	61.6	2.7	3.8
HB	1.9	93.3	0.7	1.5

Table 8: Throughput overhead (%).

to 46.7%), and HBase (232.6% to 253.1%). The memory usage increase is within 6%, with ZK from 1156.3 MB to 1219.3 MB, CS from 1209.7 MB to 1276.2 MB, HDFS from 684.8 MB to 722.7 MB, and HBase from 777.9 MB to 794.5 MB.

8 Limitations and Future Work

While T2C can generate expressive checkers from tests, our methodology relies on certain assumptions on test quality. T2C is less effective when the test: (1) does not target checking system semantics, *e.g.,* low-level data structure correctness; (2) does not include useful assertions; (3) requires complex preconditions which cannot be easily expressed or generalized. Additionally, automatically generating accurate and flexible checkers remains challenging due to limitations in existing program analysis techniques. Our precondition mutation process and the heuristics used for inferring the generalized relation constraints in checkers can also introduce inaccuracies. We plan to explore new solutions such as large language models (LLMs) to address these limitations.

T2C minimizes the safety risks of side effects by using static analysis to exclude dangerous system operations and dynamic cross-validation mechanism to filter poorly generalized checkers. However, these measures do not fully guarantee generated checkers side-effect free. To use runtime checkers in real deployment, operators could employ additional safeguards through integrating stronger isolation mechanisms (*e.g.,* sandboxing) or formal verifying generated checkers—a direction we plan to explore in future work. We also consider initially deploying T2C in testing or pre-production environments (*e.g.,* staging deployment) to be a practical way of gaining confidence before production deployment.

9 Related Work

Understanding failures in distributed systems have been a classic topic. Recent efforts pay increasing attention to complex failure patterns [25, 28, 45, 58, 69, 74, 77, 95, 96], such as fail-slow faults [50, 70, 94], metastable failures [56], and silent data corruption [85]. Lou *et al.* conduct a study on characteristics of silent semantic failures [68], which provides evidence about the prevalence and severity of such failures. Researchers have explored various testing [38, 44, 46, 64, 71, 87, 97] and verifying [31, 43, 51, 54, 59, 60, 75, 82, 91–93] techniques in software systems. Nevertheless, bugs can still occur in production due to complexity, limited resources, and scalability challenges. T2C complements existing approaches by offering a practical and low-effort approach to expose failures that escape testing.

Several solutions have been proposed to detect non-crash failures [62, 63, 76]. Panorama [57] converts components to external observers. OmegaGen [67] generates mimic checkers to detect faults within component internals. They rely on error indicators (exceptions/timeouts) to detect failures. T2C identifies silent semantic failures without explicit errors.

Many works [26, 29, 33, 66, 73, 79] provide models for developers to write runtime checks. D3S [65] allows developers to write functions that check distributed predicates as relations between states. SQCK [49] uses a declarative query language to check and repair a file system image. This manual approach is time-consuming and error-prone. These checks typically focus on well-known properties. T2C does not require excessive manual efforts. It scales to large systems and keeps the checkers up-to-date. The derived checkers capture loosely-defined semantics in modern distributed systems.

A line of work [27, 30, 35, 36, 42, 52, 61, 90] automatically mines likely invariants from software execution traces. Daikon [40] and Dinv [48] infer key state relations from test suite execution. Oathkeeper [68] infers relations between semantic-related events from test runs on buggy and patched code versions. These systems suffer from high inaccuracies due to the statistical approach. The inferred invariants have limited expressiveness to capture the diverse system semantics. T2C provides more expressive and accurate runtime checks.

PCheck [89] extracts configuration usage codes in a program as configuration checks. Ctests [81] connects the existing tests with configurations from production. ZebraConf [72] runs tests to find heterogeneous-unsafe configuration parameters. Zodiac [78] mines semantic checks from online resources for infrastructure deployment configurations. They target at detecting misconfigurations, thus address fundamentally different challenges. While Ctests also transforms a test, it focuses on configuration parameters. In contrast, T2C targets system semantics and performs more complex transformation. T2C also determines the checker precondition.

10 Conclusion

This paper explores an ambitious approach to automatically construct expressive semantic checkers for large distributed systems from existing test code. We present a study to provide insights on the feasibility. The study guides us to design and implement a practical framework T2C using static and dynamic analyses. Our evaluation of T2C on large distributed systems and real-world silent failures demonstrates its efficacy.

Acknowledgments

We thank our shepherd, Andi Quinn, and the OSDI reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-2317698, CNS-2317751, CCF-2318937, CNS-2441284, and a 4VA research grant. We thank CloudLab [39] and Google for the resources and cloud credits to support our experiments. Authors from Indonesia are supported by the MoECRT ACE Open Research program.

References

[1] CASSANDRA-14092: Max ttl of 20 years will overflow localdeletiontime. https://issues.apache.org/jira/browse/CASSANDRA-14092.

[2] CASSANDRA-14803: Rows that cross index block boundaries can cause incomplete reverse reads in some cases. https://issues.apache.org/jira/browse/CASSANDRA-14803.

[3] CASSANDRA-14873: Fix missing rows when reading 2.1 sstables with static columns in 3.0. https://issues.apache.org/jira/browse/CASSANDRA-14873.

[4] CASSANDRA-15072: Range query should return full results. https://issues.apache.org/jira/browse/CASSANDRA-15072.

[5] HBASE-21041: Memstore's heap size will be decreased to minus zero after flush. https://issues.apache.org/jira/browse/HBASE-21041.

[6] HBASE-21621: Reversed scan does not return expected number of rows. https://issues.apache.org/jira/browse/HBASE-21621.

[7] HBASE-21644: Modify table procedure runs infinitely for a table having region replication > 1. https://issues.apache.org/jira/browse/HBASE-21644.

[8] HBASE-25827: Per cell ttl tags get duplicated with increments causing tags length overflow. https://issues.apache.org/jira/browse/HBASE-25827.

[9] HBASE-28481: Prompting table already exists after failing to create table with many region replications. https://issues.apache.org/jira/browse/HBASE-28481.

[10] HDFS-14514: Snapshots should be immutable. https://issues.apache.org/jira/browse/HDFS-14514.

[11] HDFS-14699: Erasure coding should guarantee enough alive blocks. https://issues.apache.org/jira/browse/HDFS-14699.

[12] HDFS-16633: Reserved space for replicas is not released on some cases. https://issues.apache.org/jira/browse/HDFS-16633.

[13] HDFS-16942: Send error to datanode if fbr is rejected due to bad lease. https://issues.apache.org/jira/browse/HDFS-16942.

[14] Jepsen: Distributed systems safety research. https://jepsen.io/.

[15] ZK-1208: Ephemeral node should be removed after session dead. https://issues.apache.org/jira/browse/ZOOKEEPER-1208.

[16] ZK-1754: Read-only server allows to create znode. https://issues.apache.org/jira/browse/ZOOKEEPER-1754.

[17] ZK-2355: Ephemeral node should be removed after session dead. https://issues.apache.org/jira/browse/ZOOKEEPER-2355.

[18] ZK-4026: Datatree.processtxn ignores ‘opcode.create2‘ in ‘opcode.multi‘. https://issues.apache.org/jira/browse/ZOOKEEPER-4026.

[19] ZK-4325: Recursively deletion is disallowed when trying on "/". https://issues.apache.org/jira/browse/ZOOKEEPER-4325.

[20] ZK-4362: Zkdatabase.txncount logged non transactional requests. https://issues.apache.org/jira/browse/ZOOKEEPER-4362.

[21] ZK-4646: Committed txns may still be lost if followers crash after replying ack-ld but before writing txns to disk. https://issues.apache.org/jira/browse/ZOOKEEPER-4646.

[22] ZooKeeper-1208: Ephemeral node not removed after the client session is long gone. https://issues.apache.org/jira/browse/ZOOKEEPER-1208.

[23] ZooKeeper-2355: Ephemeral node is never deleted if follower fails while reading the proposal packet. https://issues.apache.org/jira/browse/ZOOKEEPER-2355.

[24] ZOOKEEPER-4837: Network issue causes ephemeral node unremoved after the session expiration.

[25] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-Tolerance, fast and slow: Exploiting failure asynchrony in distributed systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 390–408, Carlsbad, CA, October 2018. USENIX Association.

[26] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. Cloud-scale runtime verification of serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 92–107, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 4–16, New York, NY, USA, 2002. Association for Computing Machinery.

[28] George Amvrosiadis and Medha Bhadkamkar. Getting back up: Understanding how enterprise data backups fail. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 479–492, Denver, CO, June 2016. USENIX Association.

[29] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.*, 13(1):1–31, feb 1995.

[30] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 468–479, New York, NY, USA, 2014. Association for Computing Machinery.

[31] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, page 569–588, New York, NY, USA, 2007. Association for Computing Machinery.

[33] Alvin Cheung and Samuel Madden. Performance profiling with endoscope, an acquisitional software monitoring framework. *Proc. VLDB Endow.*, 1(1):42–53, aug 2008.

[34] Shigeru Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, page 313–336, Berlin, Heidelberg, 2000. Springer-Verlag.

[35] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 5–14, New York, NY, USA, 2007. Association for Computing Machinery.

[36] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 281–290, New York, NY, USA, 2008. Association for Computing Machinery.

[37] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale, 2021.

[38] David Domingo and Sudarsun Kannan. pFSCK: Accelerating file system checking and repair for modern storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 113–126. USENIX Association, February 2021.

[39] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 1–14, Renton, WA, July 2019. USENIX Association.

[40] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[41] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, December 2007.

[42] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.

[43] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 328–343, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 100–115, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 149–166, Santa Clara, CA, February 2017. USENIX Association.

[46] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 35–51, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1149–1159, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1149–1159, 2018.

[49] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 131–146, USA, 2008. USENIX Association.

[50] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Berkeley, CA, USA, 2018. USENIX Association.

[51] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 99–115. USENIX Association, November 2020.

[52] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 291–301, New York, NY, USA, 2002. Association for Computing Machinery.

[53] Klaus Havelund and Grigore Roşu. Runtime verification. *Computer Aided Verification (CAV '01) satellite workshop (ENTCS)*, 55, 2001.

[54] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[55] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[57] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16. USENIX Association, October 2018.

[58] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI, British Columbia, Canada, May 2017. ACM.

[59] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[60] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno.

Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 438–454, New York, NY, USA, 2024. Association for Computing Machinery.

[61] Choonghwan Lee, Feng Chen, and Grigore Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 591–600, New York, NY, USA, 2011. Association for Computing Machinery.

[62] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, April 2013.

[63] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, October 2011.

[64] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[65] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3s: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, page 423–437, USA, 2008. USENIX Association.

[66] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS checker: Combating bugs in distributed systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '07, Cambridge, MA, April 2007. USENIX Association.

[67] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[68] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and checking silent semantic violations in large distributed systems. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107, Carlsbad, CA, July 2022.

[69] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. Crashtuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 114–130, New York, NY, USA, 2019. Association for Computing Machinery.

[70] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. One-size-fits-none: Understanding and enhancing slow-fault tolerance in modern distributed systems. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, pages 359–378. USENIX Association, April 2025.

[71] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. Monarch: A fuzzing framework for distributed file systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 529–543, Santa Clara, CA, July 2024. USENIX Association.

[72] Sixiang Ma, Fang Zhou, Michael D. Bond, and Yang Wang. Finding heterogeneous-unsafe configuration parameters in cloud systems. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 410–425, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 365–383, New York, NY, USA, 2005. Association for Computing Machinery.

[74] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 17, USA, 2006. USENIX Association.

[75] Aurojit Panda, Mooly Sagiv, and Scott Shenker. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 30–36, New York, NY, USA, 2017. Association for Computing Machinery.

[76] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 47–62, Renton, WA, July 2019. USENIX Association.

[77] Shangshu Qian, Wen Fan, Lin Tan, and Yongle Zhang. Vicious cycles in distributed software systems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 91–103. IEEE, 2023.

[78] Yiming Qiu, Patrick Tser Jern Kon, Ryan Beckett, and Ang Chen. Unearthing semantic checks for cloud infrastructure-as-code programs. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 574–589, New York, NY, USA, 2024. Association for Computing Machinery.

[79] Andi Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. Debugging the OmniTable way. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 357–373, Carlsbad, CA, July 2022. USENIX Association.

[80] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, pages 199–215, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[81] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 735–751. USENIX Association, November 2020.

[82] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, Santa Clara, CA, July 2024. USENIX Association.

[83] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, Mississauga, Ontario, Canada, 1999. IBM Press.

[84] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 216–230, New York, NY, USA, 2023. Association for Computing Machinery.

[85] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 216–230, New York, NY, USA, 2023. Association for Computing Machinery.

[86] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.

[87] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 56–68. IEEE Press, 2023.

[88] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, page 75–82, New York, NY, USA, 2007. Association for Computing Machinery.

[89] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 619–634, Savannah, GA, November 2016. USENIX Association.

[90] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 117–132, New York, NY, USA, 2009. Association for Computing Machinery.

[91] Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. Relational network verification. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 213–227, New York, NY, USA, 2024. Association for Computing Machinery.

[92] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 485–501. USENIX Association, 2022.

[93] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 405–421. USENIX Association, 2021.

[94] Andrew Yoo, Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. Fail-slow fault tolerance needs programming support. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 228–235, New York, NY, USA, 2021. Association for Computing Machinery.

[95] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before you change: Preventing correlated failures in service updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 575–589, Santa Clara, CA, February 2020. USENIX Association.

[96] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 116–131, New York, NY, USA, 2021. Association for Computing Machinery.

[97] Li Zhong, Chengcheng Xiang, Haochen Huang, Bingyu Shen, Eric Mugnier, and Yuanyuan Zhou. Effective bug detection with unused definitions. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 720–735, New York, NY, USA, 2024. Association for Computing Machinery.

Picsou: Enabling Replicated State Machines to Communicate Efficiently

Reginald Frank, Micah Murray,
Chawinphat Tankuranand, Junseo Yoo, Ethan Xu,
Natacha Crooks, Suyash Gupta[†]*, Manos Kapritsos*[*]

University of California, Berkeley; [†] *University of Oregon;* [*] *University of Michigan*

Abstract

Replicated state machines (RSMs) cannot communicate effectively today as there is no formal framework or efficient protocol to do so. To address this issue, we introduce a new primitive, Cross-Cluster Consistent Broadcast (C3B) and present PICSOU, a practical implementation of the C3B primitive. PICSOU draws inspiration from networking and TCP to allow two RSMs to communicate with constant metadata overhead in the failure-free case and a minimal number of message resends in the case of failures. PICSOU is flexible and allows both crash fault tolerant and Byzantine fault tolerant consensus protocols to communicate. At the heart of PICSOU's good performance and generality is the concept of QUACKs (quorum acknowledgments). QUACKs allow nodes in each RSM to precisely determine when messages have definitely been received, or likely lost. Our results are promising: we obtain up to 24× better performance than prior solutions on microbenchmarks and applications, ranging from disaster recovery to data reconciliation.

1 Introduction

Many organizations today use replicated state machines (RSM) underpinned by consensus protocols to provide reliability, fault isolation, and disaster recovery. This includes key-value stores [37, 50, 79], cluster managers [25], and microservices [21, 66, 70]. These RSMs frequently need to communicate with each other in an efficient and timely manner. Etcd [37] to Etcd mirroring over Kafka, for instance, is a popular approach for disaster recovery across clusters [33]. Similarly, autonomous organizations often run their replicated key-value store locally for ease of management, but share access with other entities. For example, conversations with government agencies reveal that, for operational sovereignty, services cannot be managed across agency borders. Instead, any shared information must be communicated across RSMs and explicitly reconciled [75]. Furthermore, in the blockchain ecosystem, there is a growing push towards interoperability, which requires distinct RSMs (blockchains) to communicate [20, 90].

These examples speak of a common need: RSMs must support the ability to efficiently and reliably exchange messages with other RSMs that may or may not implement the same consensus protocol internally.

Unfortunately, existing solutions are either ad-hoc, offer vague or evolving guarantees [20], rely on a trusted third-party [92]), or require an expensive all-to-all broadcast [9, 34]. For instance, Apache Kafka, the most popular approach for exchanging data across organizations, internally relies on a third RSM for safely sharing state.

All-to-all broadcast is even more problematic: while RSMs usually run within the same datacenter, there exist many RSMs which are geographically distributed. In these cases, cross-RSM communication will take place over WAN, which offers significantly reduced bandwidth at a much higher dollar cost. This frequently causes communication to become a bottleneck.

Any system that allows RSMs to communicate should satisfy four requirements: 1) *strong guarantees*: there should be a precise and formal way to discuss RSM-RSM communication 2) *robustness under failures*: actively malicious or crashed nodes should neither affect correctness nor cause system throughput to drop [32] 3) *low-overhead in the common-case*: for efficiency, an RSM to RSM communication protocol should send a single message with constant metadata in the failure-free case 4) *generality*: arbitrary RSMs with heterogeneous sizes, communication, and fault models should be able to communicate. It should, for instance, be possible to link a Byzantine Fault Tolerant (BFT) consensus protocol with a Crash Fault Tolerant (CFT) consensus algorithm

To this effect, we first propose a new primitive, Cross-Cluster Consistent Broadcast (C3B), which can be used by two arbitrary RSMs to communicate. C3B generalizes Reliable Broadcast to guarantee that if RSM A sends m, at least one correct replica in RSM B should receive m.

We then introduce PICSOU, a practical C3B protocol that allows arbitrary RSMs with heterogeneous communication and failure models to communicate efficiently. Designing a C3B protocol that provides good performance in the failure-free case is fairly simple as a simple leader-to-leader broadcast suffices. The challenges instead arise from designing an efficient protocol that *remains robust to failures* [32]. The key to PICSOU's good and robust performance lies in observing that the C3B problem shares similar goals to TCP [80]. TCP seeks to offer reliable, ordered delivery between two hosts in a way that dynamically reacts to congestion and

anomalies in the network. To do so at low cost, TCP leverages full-duplex communication and cumulative acknowledgments (ACKs) to asynchronously detect when messages have been received. In contrast, repeated ACKs of the same message reveal message loss. PICSOU takes inspiration from these techniques and modifies them to account for the differences between C3B and TCP: 1) unlike TCP, which is exclusively designed for point-to-point messaging, PICSOU must handle many-to-many communication and disseminate knowledge of failed/successful message deliveries across many nodes, 2) PICSOU must ensure that no Byzantine participant will violate correctness or cause excessive retransmissions. In contrast, TCP does not consider malicious failures.

To address these challenges, PICSOU introduces the notion of QUACKs. A QUACK is a cumulative quorum acknowledgment for a message m. It concisely communicates the fact that all messages up to m have been reliably received by at least one correct node; *repeated* QUACKs for m indicate that the next message in the sequence was not received at the receiving RSM.

Using QUACKs in PICSOU yields multiple benefits. First, it ensures *generality*. It allows for PICSOU to seamlessly work for crash fault tolerant systems as well as for both traditional and stake-based Byzantine fault tolerant protocols. PICSOU makes no synchrony or partial synchrony assumption. Second, PICSOU's QUACK-driven implementation has low overhead. In the failure-free case, PICSOU sends each message only once, and requires only two additional counters per message. When failures do arise, PICSOU remains *robust* as its resend strategy minimizes the number of messages resent: no Byzantine node can unilaterally cause spurious message retransmissions.

Our results confirm PICSOU's strong guarantees. PICSOU allows disparate protocols such as PBFT [28], Raft [69] and Algorand [40] to communicate. On two real-world applications, Etcd Disaster Recovery [33] and a data reconciliation application [75], PICSOU achieves 2× better performance than Kafka. In our microbenchmarks, when consensus is not the bottleneck, PICSOU achieves 3.2× better performance than a traditional All-to-All broadcast for small networks (4 nodes), and up to 24× for large networks (19 nodes). In summary, this paper makes the following contributions:

1. We introduce Cross-Cluster Consistent Broadcast primitive, which allows for two RSMs to communicate robustly and efficiently.
2. We present PICSOU, a practical C3B protocol. Key to PICSOU's good performance is the use of QUACKs (cumulative quorum acknowledgments), which precisely determines when messages have definitely been received, or likely lost.
3. We evaluate PICSOU on realistic workloads, showing that it can successfully allow disparate RSMs to communicate more effectively that prior solutions.

2 Formalising the C3B primitive

We first introduce and formalize the C3B primitive. C3B is the blueprint for any communication protocol between RSMs and should be sufficiently general to support various communication and failure models.

2.1 System Model

We first discuss the necessary formalism. Consider a pair of communicating RSMS. For the sake of exposition, we denote the sending RSM as $\mathcal{R}_s$ and the receiving RSM as $\mathcal{R}_r$. In practice, communication between these RSMs is full-duplex: both RSMs can send and receive messages.

Most modern RSMs are either *crash fault tolerant* (they guarantee consensus when up to $\mathbf{f}$ nodes crash) or *Byzantine fault tolerant* (they guarantee consensus when up to $\mathbf{f}$ nodes behave arbitrarily). In line with PICSOU's stated generality and efficiency goals, we adopt the *UpRight* failure model [31]. It allows us to consider Byzantine nodes and crashed nodes in a unified model, letting us design a system that optimizes for each type of failure. In the UpRight failure model, Byzantine nodes may exhibit *commission failures*; they may deviate from the protocol. All other faulty nodes may suffer from *omission failures* only: they follow the protocol but may fail to send/receive messages. Crashed nodes, for instance, suffer from permanent omission failures once crashed. Correct nodes, by definition, never fail[1]. In this setup, each RSM consists of $\mathbf{n}$ replicas. We denote the j-th replica at the i-th RSM as R_{ij} (where i is either the sender or the receiver RSM). Each RSM interacts with a set of clients, of which arbitrarily many can be faulty.

We say that an RSM is safe despite up to $\mathbf{r}$ commission failures and live despite up to $\mathbf{u}$ failures of any kind. For example, using the UpRight model, we can describe traditional BFT and CFT RSMs using just one equation: $2\mathbf{u}+\mathbf{r}+1$; Setting $\mathbf{u}=\mathbf{r}=\mathbf{f}$ yields a $3\mathbf{f}+1$ BFT RSM and setting $\mathbf{r}=0$ yields a $2\mathbf{f}+1$ CFT RSM. Safety and liveness of any RSM are defined as follows:

Safety. If two correct replicas R_{i1} and R_{i2} in RSM $\mathcal{R}_i$ commit transactions T and T' at sequence number k, then $T=T'$.

Liveness. If a client sends a transaction T to RSM $\mathcal{R}_i$, correct replicas in $\mathcal{R}_i$ will eventually commit T.

Note that we make no assumptions about the communication model of the underlying RSM. We only assume messages are eventually delivered and that the receiving RSM $\mathcal{R}_r$ can verify whether a transaction was in fact committed by the sender RSM $\mathcal{R}_s$ (more details in Section 3).

[1] This is of course a simplification, necessary to say anything about an RSM. All practical systems assume that this $\mathbf{f}$ stays true at any given point in time, and that the set of failed nodes only changes. It allows making statements such as "$\mathbf{f}+1$ votes ensure that at least one correct node participated".

We generalize our system model to support *shares*. The notion of share is used in stake-based BFT consensus protocols where the value of a share determines the *decision-making power of a replica* [42]. For a replica R_{ij} in RSM $\mathcal{R}_i$, we write δ_{ij} to represent the replica's stake; the total amount of share in RSM $\mathcal{R}_i$ is then $\sum_{l=1}^{|\mathbf{n}_i|} \delta_{il}$. A stake-based consensus algorithm is safe as long as replicas totaling no more than $\mathbf{r}_i$ shares deviate from the protocol. The system is live as long as no more than $\mathbf{u}_i$ shares fail to send/receive messages Traditional CFT and BFT algorithms simply set all shares equal to one.

Adversary Model. We assume the existence of a standard adversary which can corrupt arbitrary nodes, delay and reorder messages but cannot break cryptographic primitives. As is standard, we assume that reconfigurations are possible [7, 38] and that there exists a mechanism for an RSM to reliably learn of the new configuration and/or share assignments. We provide more detail on how such a mechanism can be implemented in Section 4.4

2.2 Cross-Cluster Consistent Broadcast

The C3B primitive enables efficient and reliable communication between a sender RSM and a receiver RSM.

To formalize C3B, we first need to define two new communication primitives that express exchanging messages between RSMs. These operations are at a coarser granularity than the traditional *send* and *receive* operations, which define the operations performed by a specific node or replica.

Transmit. If a correct replica in $\mathcal{R}_s$ invokes C3B on message m, we say that RSM $\mathcal{R}_s$ *transmits* message m to RSM $\mathcal{R}_r$.

Deliver. If a correct replica from $\mathcal{R}_r$ outputs message m, we say that RSM $\mathcal{R}_r$ *delivers* message m from $\mathcal{R}_s$.

We can then define the two correctness properties that all C3B implementations must satisfy:

Eventual Delivery. If RSM $\mathcal{R}_s$ transmits message m to RSM $\mathcal{R}_r$, then $\mathcal{R}_r$ will eventually deliver m.

Integrity. For every message m, an RSM $\mathcal{R}_r$ delivers m from $\mathcal{R}_s$ if and only if $\mathcal{R}_s$ transmitted m to $\mathcal{R}_r$.

Note that the deliver operation requires only that one correct node receives the message, not that all correct nodes receive it. This is by design: our goal is to make the C3B condition as flexible as possible to suit application needs. In practice, it is easy to strengthen this condition to either guarantee delivery to all nodes or to establish ordering between RSMs, as correct nodes in the receiving RSM can simply broadcast or invoke consensus on delivered messages. In the name of generality, the C3B primitive talks about a single message only, and does not worry about ordering across messages. PICSOU, nonetheless, uses knowledge of ordering for efficiency, as we describe next.

3 Design Overview

PICSOU *efficiently* implements the C3B primitive while remaining *general* and *robust* under failures. The design of PICSOU is centered around three pillars:

(P1) **Efficiency.** In the (common) failure-free case, where messages are received in a timely fashion, PICSOU should only send a single copy of each message across RSMs, and no more than $O(n)$ copies within a cluster. Any additional metadata sent as part of PICSOU should have constant size.

(P2) **Generality.** PICSOU should support RSMs of arbitrary sizes, with diverse failure models and communication models, including crash and Byzantine faults as well as synchronous and asynchronous networks. The protocol logic must also work well for both traditional BFT systems and newer Proof-of-Stake protocols, where a replica's share determines the weight that its vote carries.

(P3) **Robustness.** PICSOU should remain robust to failures. Crashed or malicious replicas should have minimal impact on performance [32]. There is a tension here: while the protocol should aggressively resend dropped messages to minimize latency, Byzantine nodes should not cause correct nodes to resend messages, which can spuriously hurt throughput.

Much like TCP flows, communication between RSMs is *streaming*, *long-running* and often *full-duplex*. PICSOU thus draws inspiration from TCP's approach to congestion control and message loss to guarantee efficiency and robustness.

Two ideas are central to TCP's good performance: 1) leveraging full-duplex communication and 2) cumulative ACKing. In TCP, nodes simultaneously exchange messages, and TCP leverages this bidirectional communication to piggyback acknowledgments onto messages and minimize bandwidth requirements. Cumulative ACKing then keeps these acknowledgments small: with a single counter k, a receiver informs a sender that it has received all packets with sequence number up to k. Receiving a repeated counter with value k instead informs the sender that the packet with sequence number $k+1$ is either lost or delayed.

For *efficiency*, PICSOU also leverages full-duplex communication and cumulative ACKing. However, PICSOU must also handle many-to-many communication as each RSM consists of multiple replicas. For *generality*, PICSOU must support crashed nodes, Byzantine nodes, as well as Byzantine nodes whose stake determines their voting power; PICSOU makes use of the *UpRight* fault model to simultaneously handle crash and Byzantine nodes, and leverages the mathematics of *apportionment* to work seamlessly with staked-based systems. For *robustness*, PICSOU must ensure that replicas cannot trigger spurious message retransmissions. PICSOU uses QUACKs to determine when a message has definitely been received or is likely lost.

Overview. PICSOU's protocol logic can be divided into the following logical components.

(1) *Consensus*: On either side of PICSOU lies a replicated state machine. Each RSM receives requests from clients and runs a consensus protocol that commit these requests on each RSM replica. PICSOU assumes two properties of consensus to guarantee correctness: first, all replicas eventually receive all messages, and second, all replicas agree on the content of each slot in the log.

(2) *Invoking* PICSOU: If an RSM wants to transmit a request, each replica forwards the committed request to the co-located PICSOU library. RSMs are not required to forward every committed message to PICSOU. This step is application-specific. For example, if two organizations only share a subset of their data, the RSM will only transmit messages that touch these particular objects.

(3) *Transmitting a message*: PICSOU sends the message on behalf of the sending RSM. In line with our stated efficiency goals, PICSOU ensures that, in the absence of failures and during periods of synchrony, a *single sender* forwards each request to a *single replica* in the receiving RSM. To minimize the risk of repeated failures caused by a faulty sender or receiver, PICSOU carefully rotates sender-receiver pairs. In doing so, it ensures that every sender will eventually communicate with a correct receiver and vice-versa.

(4) *Detecting successful or failed sends:* PICSOU must quickly determine whether a message has definitely been received (and thus can be garbage collected) or has definitely been dropped or delayed (distinguishing between message drops and delays is not possible in an asynchronous system). Failure detection must be accurate to prevent Byzantine nodes from causing spurious re-transmissions; it must be efficient and should not require additional communication between nodes. PICSOU adapts TCP's cumulative acknowledgment approach to detect when messages have been received or dropped, even when malicious replicas can lie. These acknowledgments are piggybacked on incoming messages, thus minimizing overhead.

(5) *Retransmissions*: When the protocol detects that a message has been dropped (with high confidence), PICSOU intelligently chooses the node responsible for resending the message. Unfortunately, concurrent node failures can cause multiple messages to be dropped simultaneously. To address this issue, PICSOU includes (constant size) information about which messages have been lost, allowing the protocol to recover multiple dropped messages in parallel. Traditional TCP can eschew this constraint as it assumes failures are rare and considers point-to-point communication only.

Correctness We defer a full description of correctness and proofs to Appendix A.1 and A.2 [1].

4 Protocol Design

We now describe each component of the protocol: transmitting a message, detecting successful/failed sends, and retransmissions. For clarity of exposition, we describe the protocol as consisting of a sender RSM $\mathcal{R}_s$ and a receiver RSM $\mathcal{R}_r$ proceeding in synchronous timesteps. In practice, nodes operate independently and act as both the sender and the receiver. We start by describing PICSOU's behavior in the common case (§4.1) before considering failures (§4.2). We add support for stake in §5. We include protocol pseudocode in Appendix A.3 [1].

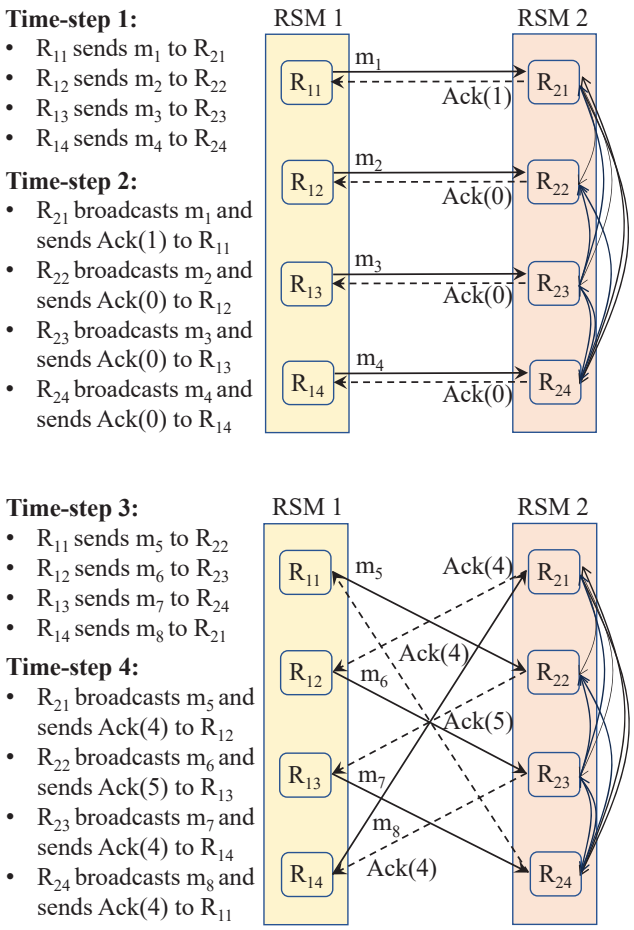
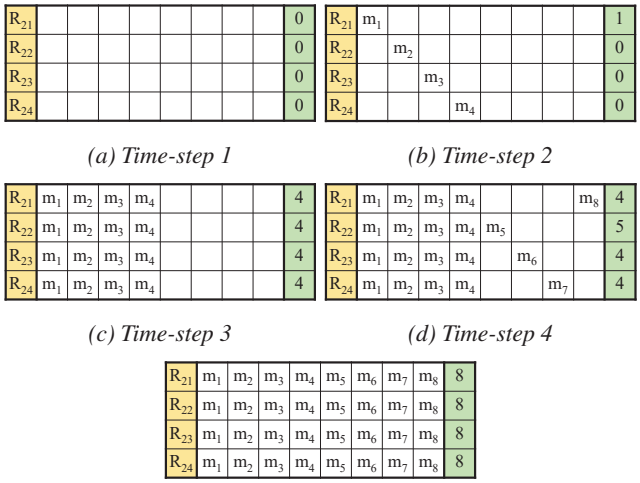


Figure 1: Example failure-free run in PICSOU

4.1 Failure-free behavior

Sending a message. PICSOU's send logic has three goals: 1) minimize the number of nodes sending the same message 2) maximize the chances that a message will be received quickly 3) asynchronously disseminate knowledge of received messages to other nodes. PICSOU achieves these goals by round-robin partitioning the set of requests across all sending RSM replicas and rotating receiver nodes every round.

By definition, each replica in an RSM contains a log of committed requests, a subset of which should be transmitted to the other RSM. PICSOU assumes that each request transmitted through PICSOU is of the form $\langle m,k,k' \rangle_{Q_s}$, where m is a request committed at sequence number k by a quorum of replicas in RSM $\mathcal{R}_s$. Each protocol sets a specific threshold t above which the request has acquired sufficiently many



(a) Time-step 1 *(b) Time-step 2*

(c) Time-step 3 *(d) Time-step 4*

(e) Time-step 5

Figure 2: Receiver's view of events in Figure 1 in time-steps.

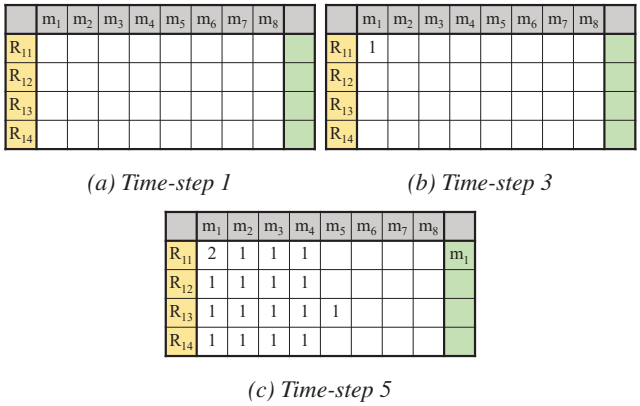
signatures Q_s to be deemed committed. k' is an optional sequence number that denotes the position of the message in the stream of messages that will be transmitted through PICSOU. k' must be sequentially increasing; $k' = \perp$ indicates that the message should not be transmitted. Including both k and k' allows $\mathcal{R}_s$ to filter which messages will be transmitted to $\mathcal{R}_r$.

PICSOU evenly partitions the task of transmitting committed messages across all replicas such that each message is sent by a single node: replica R_{sl} sends messages with sequence number $(k' \bmod \mathbf{n}_s \equiv l)$. Additionally, each sender rotates receivers on every send: if RSM $\mathcal{R}_r$ has size $\mathbf{n}_r$ and replica R_{sl} last sent to replica R_{rj}, then R_{sl} will next send to R_{rJ}, where $J \equiv j+1 \bmod \mathbf{n}_r$. Node IDs themselves are assigned by PICSOU using a verifiable source of randomness [40] such that malicious nodes cannot choose specific positions in the rotation. Note that, as is standard in TCP, we allow senders to transmit a window of messages in parallel.

Receiving a message Upon receiving a message, the j-th replica R_{rj} checks that the message $\langle m, k, k' \rangle_{Q_s}$ is valid (the message has provably been committed by the sender RSM) and if so, broadcasts it to the other nodes in its RSM. Importantly, the receiving node does not need to recommit the message. It can simply apply it to its local state as mandated by the application logic.

Rotating sender-receiver pairs in this way guarantees that every pair of replicas will eventually exchange messages and ensures that (1) information about the state of each node is propagated to every other node in the system, and (2) no sender is continuously sending to a faulty replica (or vice-versa). This process is also essential to bounding the number of retransmissions needed with failures (§4.2).

We illustrate PICSOU's logic in Figure 1. For clarity of exposition, we assume that 1) in each time-step, each replica completes all relevant tasks in parallel, 2) all sent messages are received in the next time-step, and 3) only one RSM sends, the other only acks. In our implementation, acks are piggybacked on messages. Consider a system with $\mathbf{n}_s = \mathbf{n}_r = 4$ replicas ($\mathbf{u} = \mathbf{r} = 1$). In time-step 1, replicas R_{11}, R_{12}, R_{13}, and R_{14} of RSM_1 send messages m_1, m_2, m_3, and m_4 to receivers R_{21}, R_{22}, R_{23}, and R_{24}, respectively. In time-step 2, these receivers internally broadcast these messages to the other nodes in their RSM. Concurrently, R_{21}, R_{22}, R_{23}, and R_{24} acknowledge receipt of these messages and send $\text{ACK}(1)$, $\text{ACK}(0)$, $\text{ACK}(0)$, and $\text{ACK}(0)$ to senders R_{11}, R_{12}, R_{13}, and R_{14}. We discuss acknowledgments later in the section. In time-step 3, R_{11}, R_{12}, R_{13}, and R_{14} rotate receivers and send messages m_5, m_6, m_7, and m_8 to R_{22}, R_{23}, R_{24}, and R_{21}. In time-step 4, receivers once again broadcast the received messages in their RSM. Simultaneously, R_{21}, R_{22}, R_{23}, and R_{24} rotate receivers for their acknowledgements, and send $\text{ACK}(4)$, $\text{ACK}(5)$, $\text{ACK}(4)$, and $\text{ACK}(4)$ to senders R_{12}, R_{13}, R_{14}, and R_{11}.



(a) Time-step 1 *(b) Time-step 3*

(c) Time-step 5

Figure 3: Sender's view of events in Figure 1 in time-steps.

Detecting successful sends. To guarantee correctness, committed messages must eventually be received by a correct node in the receiving RSM. Every node in the sending RSM (not just the sender) must thus learn whether a message has *definitely* been received by a correct node. This is necessary to preclude correct nodes from unnecessarily resending messages. There are three primary challenges: (1) malicious nodes may lie about the set of messages received, (2) for efficiency, PICSOU should not require nodes within an RSM to exchange information beyond the necessary message broadcast, and (3) any additional metadata should be small. PICSOU realizes these goals through *cumulative quorum acknowledgments* (or QUACKs). A cumulative quorum acknowledgment with value k proves to the sending RSM that all messages with sequence number up to k have been received by at least one correct replica. More specifically, each replica, upon receiving a message with sequence number k, inserts it into a sorted list containing all previously received messages. The replica then identifies the highest message m_p in the list for which all messages with a smaller sequence number have been received. It then crafts an acknowledgment $\text{ACK}(p)$ that cumulatively acknowledges receipt of messages 1 to p. PICSOU takes advantage of the full-duplex nature of

the protocol to piggyback these acknowledgments onto the messages that the receiving RSM is itself sending to the sending RSM. If no such message exists, the RSM sends a no-op.

On the sender side, each replica eventually receives messages and acknowledgments from all $\mathbf{n}_r$ receiving replicas, thanks to PICSOU's round-robin strategy. Each replica maintains an $\mathbf{n}_r$ sized array that summarizes the highest acknowledgment received from each replica of receiving RSM. A message m_p is QUACKed if at least $\mathbf{u}_r + 1$ replicas have acknowledged receipt of all messages up to p. As there are only $\mathbf{u}_r$ failed replicas, one of these replicas must be correct. We thus have the guarantee that this correct replica will broadcast the message to all other remaining correct replicas. Note that PICSOU additionally uses MACs to safely transmit ACKs when configured to handle Byzantine failures ($\mathbf{r} > 0$).

Example. Continuing with our example in Figure 1, we highlight the protocol logic for both the receiver RSM (Figure 2) and the sender RSM (Figure 3). Each row in Figure 2 describes a sorted list of messages received at each replica, with the last column denoting the highest cumulative acknowledgment for this node. At time-step 1, all lists are empty and cumulative acknowledgment values are all set to 0 (2a). At time-step 2, receivers R_{21}, R_{22}, and R_{23} (2b) store messages m_1, m_2 and m_3. R_{21}'s cumulative acknowledgment counter thus increases to 1, while others stay at 0 as they are still missing m_1. R_{21}, R_{22}, and R_{23} thus send $\text{ACK}(1)$, $\text{ACK}(0)$ and $\text{ACK}(0)$ to the sender RSM. At time-step 3, each receiver, thanks to the internal broadcast mechanism, receives all four messages. All cumulative acknowledgment counters thus go to 4 (2c). By time-step 4, receivers R_{21}, R_{22}, and R_{23} have all received m_8, m_5 and m_6. R_{22} has received messages m_1 to m_5, and thus updates its cumulative acknowledgment to 5. In contrast, R_{21} and R_{23} have received messages m_8 and m_6 respectively, they are missing m_5 and thus cannot yet update their cumulative acknowledgment counter. R_{21}, R_{22}, and R_{23} send $\text{ACK}(4)$, $\text{ACK}(5)$ and $\text{ACK}(4)$ back to the sending RSM. Finally, at time-step 5, the internal broadcast mechanism disseminates all these messages; each replica can update its cumulative acknowledgment to 8.

Now consider the sender-side logic (Figure 3), which processes these cumulative acknowledgments and determines when a QUACK has formed. Recall that a message is QUACKed at a replica if this replica receives $\mathbf{u} + 1 = 2$ acknowledgments for m. Initially, all QUACK counters are empty (3a). At time-step 3 (3b), R_{11} records that it has received an acknowledgment for m_1 ($\text{ACK}(1)$) from R_{21}. At time-step 5, (3c), R_{11} receives $\text{ACK}(4)$ from R_{24}. It updates its local array, indicating that it has received acknowledgments for m_1 by two unique nodes, and marks m_1 as QUACKed. It also records the received acks for m_2, m_3, and m_4. Similarly, all other replicas indicate that they have received an ack for m_1, m_2, m_3, and m_4. Additionally, R_{13} indicates it received an ack for m_5 as it received $\text{ACK}(5)$.

Summary The joint techniques of full-duplex communication, cumulative acking, and rotation of sender/receiver pairs allow PICSOU to ensure that all RSM replicas eventually learn which committed requests have been delivered. The protocol achieves this with only two additional counters and with no additional communication between replicas of an RSM beyond the necessary broadcast.

4.2 Handling Failures

Faulty replicas can: 1) fail to send or broadcast messages sends and broadcasts, 2) send invalid messages to DDoS the network, 3) collude to repeatedly drop messages, and 4) send incorrect acknowledgments to break correctness or trigger spurious retransmissions. PICSOU must effectively handle these failures without sacrificing correctness or performance. To this effect, PICSOU must quickly and reliably detect *when* a message has *definitely* been dropped or delayed and quickly retransmit it. The system must do so without any additional communication beyond resending the message itself.

Detecting failed sends. The protocol once again leverages QUACKs to detect failed sends. Recall that all sender replicas eventually obtain a QUACK for every message that has *definitely* been delivered. One can instead leverage duplicate QUACKs to learn when a correct replica has *not* received a specific message. In more detail, let us assume that a QUACK for message m_k has formed at R_{sl}. This QUACK indicates that at least $\mathbf{u}+1$ (at least one correct) replicas have received every message up to message m with sequence number k. A duplicate acknowledgment $\text{ACK}(k)$ from one of these replicas implies that this replica claims not to have received the message at sequence number $k+1$. Once a duplicate QUACK forms for the k-th message at replica R_{sl}, R_{sl} learns that a correct replica is complaining about missing message $(k+1)$, and thus that the message has legitimately been lost or delayed. All other replicas of the sending RSM $\mathcal{R}_s$ will eventually receive a duplicate QUACK and thus detect the failed exchange. The use of the UpRight failure model, which distinguishes actively malicious failures $\mathbf{r}$ from all other failures, allows us to reduce the size of the duplicate QUACK: while the initial QUACK is of size $\mathbf{u}+1$, duplicate QUACKs must be of size $\mathbf{r}+1$ as they must be large enough to preclude actively malicious nodes from triggering spurious resends. In a system with only crash failures (when $\mathbf{r}=0$), a single duplicate ACK() is sufficient to trigger a message resend: nodes may crash but do not lie.

Retransmitting the dropped message. Upon detecting a failed send, the message must be quickly retransmitted. Just as a single replica was responsible for sending the initial message, PICSOU ensures that a single replica is "elected" as the re-transmitter. It does so *without* requiring additional communication between replicas. The protocol logic hinges on three observations: 1) all correct replicas know about all the messages that must be transmitted (by definition of an RSM) and know who initially sent the message, 2) all

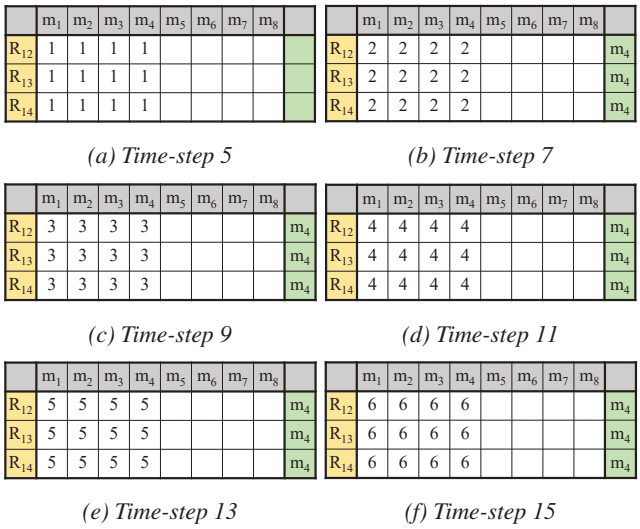


Figure 4: Sender's view of events. R_{11} *fails after TS 2 in Figure 1.*

correct replicas eventually learn about which messages have been QUACKed, and 3) the number of repeated QUACKs indicates the number of failed retransmissions. PICSOU uses this information to compute the ID of the re-transmitter as: $sender_{new} = (sender_{original} + \#_{retransmit}) \bmod \mathbf{n}_s$. Each correct replica computes this function and retransmits the message if its ID matches $sender_{new}$. Each retransmission round thus has a single sender.

To illustrate, consider once again our initial example (Figure 1), but this time, let us assume that sender replica R_{11} fails in time-step 2, after sending message m_1 but before sending messages m_5 and m_9. As a result, no receiver receives these messages. In Figure 4, we time-step through this failure scenario. For simplicity, we assume that the receiving RSM sends periodic acks every time-step. As before, all non-failed replicas of RSM_1 receive their first ACK(4) in time-step 5 (Figure 1); In time-step 7, all replicas receive a second ACK(4) message, from a different node, allowing them to mark messages m_1 to m_4 as QUACKed. They continue receiving ACK(4) from *distinct* replicas in time-step 9 and 11. Receivers cannot acknowledge any message greater than m_4 as they are yet to receive m_5. In time-step 13, the senders receive their first duplicate ACK(4) message. By the end of time-step 15, the senders have received at least $\mathbf{r}_r + 1 = 2$ duplicate ACK(4) messages, confirming that m_5 is missing. R_{12} proceeds to resend m_5.

The pitfalls of sequential recovery. Unlike traditional TCP in which message drops are not adversarial, faulty replicas can carefully select which messages to drop. For instance, in a $\mathbf{n} = 2\mathbf{u} + \mathbf{r} + 1$ setup with $\mathbf{u} = \mathbf{r} = 1$, if a node omits all received messages, every fourth message will need to be resent. In this setup, PICSOU can hit a throughput bottleneck. A QUACK conveys information about the *lowest* message that has been dropped by the system, but says nothing about later messages. This approach is optimal metadata-wise but serializes recovery: if messages m_i, m_{i+4}, m_{i+8}, etc. have all been dropped, resending m_{i+8} first requires detecting the failed send of message m_i, retransmitting m_i, QUACK m_i, before repeating the same process for m_{i+4}. Only then can the failed send of m_{i+8} be handled.

Parallel Cumulative Acknowledgments. To address this issue, we must augment our cumulative acknowledgments with a limited form of selective repeat [93]. Each receiver sends both a cumulative acknowledgment and a list summarizing the delivery status of up to ϕ messages past the sent cumulative acknowledgment. The cumulative acknowledgment counter concisely summarizes the set of contiguous messages received so far. The ϕ-list instead captures any "in-flight" missing messages. Sending ϕ-lists over the network is efficient as the delivery status of each message takes at most one bit to encode. This list can further be reduced with techniques such as compression or bloom filters.

Sender replicas can now, concurrently, form QUACKs for ϕ concurrent messages and thus retransmit ϕ messages in parallel. This reduces latency without resorting to eager message resends. The maximum size of ϕ-lists is an experiment-specific parameter. The actual number of elements in a ϕ-list depends on the number of in-flight messages at the time of sending a cumulative acknowledgment.

Analysis During periods of synchrony (when messages are not dropped or delayed by the network), PICSOU retransmits messages at most $\mathbf{u}_s + \mathbf{u}_r + 1$ times. This limitation is fundamental to all C3B protocols (Lemma 1 in Appendix [1]). The number of resends can become a concern for latency if the number of failures is large. In practice, however, the probability of actually hitting this bound is small. Intuitively, in a CFT or BFT system, each node is more likely than not to be correct. As such, the probability of continually selecting incorrect nodes in every sender-receiver pair decreases exponentially every retry. One can use this reasoning to provide strong bounds on the maximum number of retries when the network is well-behaved. We prove, for instance (in Appendix A.2 [1]) that PICSOU needs to resend a message at most eight times to ensure that a message be delivered with 99% probability, and at most 72 times to ensure a $100 - 10^{-9}$% success probability.

4.3 Garbage Collection

At first glance, garbage collecting messages in PICSOU appears straightforward. The sending RSM, upon receiving a QUACK for m, can garbage collect m as the message has been received by a correct replica. Unfortunately, this approach can lead to scenarios in which PICSOU stalls. Consider, for instance, an execution in which sender R_{sl} sends a message m_k (at sequence number k) to replica R_{rj} of RSM $\mathcal{R}_r$. Now, consider the case in which R_{rj} is faulty and broadcasts m_k to precisely $\mathbf{u}_r + 1$ replicas, $\mathbf{u}_r$ of which are faulty. These replicas reply to the sender RSM that m has been successfully received, allowing for a QUACK to form at the sender, and for message m to be garbage collected. Unfortunately, if these

$\mathbf{u}_r$ replicas then stop participating in the protocol, no QUACK will ever form for any message with sequence number greater than k (only one correct replica has seen m). Instead, the sending RSM will receive repeated duplicate acknowledgments for m, a message which it has already garbage collected!

We must consequently modify the garbage collection algorithm slightly. If a sending replica ever receives a duplicate QUACK for message $m_{k'}$ where $k' < k$ *after* having quacked and garbage collected message m_k, it includes, as additional metadata, the sequence number k of its highest quacked message. This information conveys to the receiving RSM that all messages up until k (included) have been received by *some* correct node in the receiving RSM, but not necessarily the same one. Replicas in the receiving RSM, after having received $\mathbf{r}_s + 1$ such messages (ensuring that at least one correct node is in the set), can then either (1) advance their cumulative acknowledgment counter to k and mark message m as received, or (2) obtain m from other replicas in the RSM. Only then can m be garbage collected. We offer both strategies in PICSOU.

4.4 Reconfiguration

PICSOU assumes that reconfigurations are possible but rare. It assumes that there exists a service indicating the set of nodes associated with each configuration. This is standard practice in the literature [7, 37, 38]. Knowledge of membership is either maintained internally in the RSM [87] or using an external configuration service [37, 87]. Most existing state-of-the-art blockchain systems [3, 10, 40, 51, 59, 67, 75, 83, 94] also require known node membership. To deal with churn and scale, these systems work in epochs where it is assumed both node membership and relative stake are both publicly known and fixed. PICSOU piggybacks on this assumption. PICSOU then only needs to ensure that the set of ACKs received for a particular message all match the same view and that the relevant $(\mathbf{u}+1/\mathbf{r}+1)$ threshold has been reached (for that view).

Messages acknowledged as delivered before a reconfiguration occurs do not need to be resent. Reconfiguration in an RSM, by definition, preserves any state across configurations. Messages not acknowledged as delivered before the reconfiguration begins must be resent as they may or may not have persisted. After reconfiguration completes, PICSOU simply resends messages for which it did not receive a quorum of acknowledgments in the prior configuration.

5 Weighted RSMs – Stake

The current description of the protocol assumes that replicas have equal weight in the system. When considering proof-of-stake systems like Algorand, each replica can instead hold differing amounts of *stake* or *shares* in the system. We write δ_{ij} for the share of R_{ij}; the total amount of stake in RSM $\mathcal{R}_i$ is then $\mathbf{\Delta_i} = \sum_{l=1}^{\mathbf{n}_i} \delta_{il}$. The RSM is safe as long as replicas totalling no more than $\mathbf{r}_i$ shares deviate from the protocol; the RSM is live as long as replicas totalling no more than $\mathbf{u}_i$ shares omit messages. The existence of stakes changes: (1) when a replica can establish a QUACK, and (2) to whom a particular message must be sent.

5.1 Weighted QUACK

It is straightforward to modify QUACKs to deal with stake. Each cumulative acknowledgment message simply becomes weighted. The acknowledgment message from replica R_{il} with share δ_{il} has a weight δ_{il} and a QUACK forms for message m when the total weight of the cumulative QUACK for m from RSM $\mathcal{R}_i$ is equal to $\mathbf{u}_i + 1$.

5.2 Sending a message

Identifying the appropriate sender-receiver pair for sending a message requires more care. Traditional BFT systems couple voting power, physical node and computation power. This is no longer the case with stake: different nodes can have arbitrarily different stakes. This problem is compounded by the fact that stake is unbounded and often in the billions [40]. A single physical node can effectively carry both arbitrarily large or arbitrarily small stake.

We want to ensure that we maintain the same correctness and performance guarantees as in non-staked systems. Unfortunately, the round-robin approach we described in §4.1 no longer works well. Consider for instance a system with $\mathbf{n}_i = 1000$ total stake, spread over two machines. R_{i1} is Byzantine and has $\delta_1 = \mathbf{u}_i = 333$, while R_{i2} has $\delta_2 = 667$. Using round-robin across these replicas disproportionately favors R_{i1} which represents only 33.3% of the shares in the system, yet is tasked with sending/receiving half the total messages. We must thus skew choosing sender-receiver pairs towards nodes with higher stake. To highlight the challenges involved, we first sketch two strawmen designs:

- *Version 1: Skewed Round-Robin.* The most straightforward approach is to have replica R_{il} with stake δ_{il} use round-robin scheduling to send δ_{il} messages on its turn. This is, eventually, completely fair since all nodes send precisely as many messages as they have stake in the system. Unfortunately, this solution suffers from very poor performance under failure as it has *no parallelism*: if stake is in the order of billions in the system, a single faulty node may fail to send large contiguous portions of the message stream, triggering long message delivery delays. Rounding stake is unfortunately not an option: as stake is unbounded, each physical node can, in effect, have infinitely small (or arbitrarily large) stake in the system. One physical node can have $\delta_l = 1$ while another has $\delta_l = 1 \times 10^9$. Rounding errors weaken liveness as more retransmissions may be needed to identify a correct sender-receiver pair.
- *Version 2: Lottery Scheduling.* For our next attempt, we consider lottery scheduling, a probabilistic scheduling algorithm [89]. Each node is allocated a number of tickets according to its stake; the scheduler then draws two random tickets to

choose the next sender and the next receiver. Lottery scheduling addresses the parallelism concern mentioned above. Over long periods of time, the protocol is completely fair, and each sender-receiver sends/receives according to its stake. Unfortunately, due to the randomized nature of the protocol, over short periods of time, the proportion of sender and receiver pairs chosen may skew significantly from their shares in the system.

Dynamic Sharewise Scheduler. Our solution must (1) offer good parallelism; trustworthy replicas should be able to send messages in a bounded unit of time, (2) ensure fairness over both short and long periods; each node should send messages proportional to its shares, and (3) tolerate arbitrary stake values. These properties are exactly those that the Linux Completely Fair Scheduler (CFS) seeks to enforce. CFS defines a configurable time quantum during which each process is guaranteed to be scheduled; each process then gets CPU time proportional to its priority.

Our *dynamic sharewise scheduler* (DSS) adopts a similar strategy with one key modification. As stake is unbounded, DSS cannot guarantee as easily as CFS that all nodes will send a message within a fixed time period t. Instead, DSS maximizes the following objective: given a fixed time period t, how can PICSOU schedule sender-receiver pairs such that each node sends/receive messages *proportionally* to its shares. While this may appear straightforward, the ability for nodes to have arbitrarily large (or small) stake makes reasoning about proportionality challenging. DSS turns to the mathematics of *apportionment* to handle this issue [19, 81]. Note that PICSOU uses DSS to identify both senders and receivers in the same way. For simplicity, we thus discuss apportionment from the perspective of senders only.

Apportionment is used to fairly divide a finite resource between parties with different entitlements or weights[2]. Formally, an apportionment method M defines a multivalued function $M(\vec{t},q)$. Here, $\vec{t}$ represents the entitlement of node R_{il}, that is the amount of messages that it should send or receive. In our case, this corresponds to its stake $\vec{t}_{il} = \delta_{il}$. q denotes the total number of messages that can be sent in the specified time quantum t. DSS makes use of Hamilton's method of apportionment [19, 81], which proceeds in four steps:

- First, DSS finds the standard divisor (SD_i), the ratio of the total amount of stake over the number of messages in a quantum, $SD_i = \frac{\Delta_i}{q}$. Intuitively, this defines how much stake must "back" each message.
- Next, DSS computes the standard quota (SQ_{il}) for each node R_{il}, $SQ_{il} = \frac{\delta_{il}}{SD_i}$, which indicates how many messages each replica should send. As this number may not be a whole number, DSS also computes the matching *lower quota* (LQ_{il}), which takes the floor of the standard quota. The difference between the standard quota and the lower quota is called the penalty ratio PR_{il}.

[2]It is, for instance, used to assign the number of seats per state in the US House of Representatives.

DSS	Stake	q	δ_0	δ_1	δ_2	δ_3	c_0	c_1	c_2	c_3
d_1	100	100	25	25	25	25	25	25	25	25
d_2	1000	100	250	250	250	250	25	25	25	25
d_3	1000	100	214	262	262	262	22	26	26	26
d_4	100	10	97	1	1	1	10	0	0	0

Figure 5: Apportionment Example. c_0,...c_3 refers to the number of messages that must be sent (or received) by each node per quanta

- DSS adds up these lower quotas to find the number of messages that will be sent $q_{whole} = \sum_l^{\mathbf{n}_i} LQ_{il}$, without worrying about any unfairness introduced by rounding.
- If $q_{whole} < q$, that is if there is still space to send additional messages, DSS decides to increment the allocation of each R_{il}, in decreasing order of penalty ratio PR_{il}.

Worked Example. Intuitively, the algorithm described above splits messages fairly across nodes while minimizing the degree of imbalance introduced by the need to round stake up or down. Consider for instance the stake distribution and message quanta in Figure 5. The first two scenarios are straightforward as each replica has equal amounts of stake. In both settings, running Hamilton methods, with a SD of 1 in d_1 and of 10 in d_2 reveals that each node should send 25 messages. d_3 highlights where apportionment shines. In this example, stake is not distributed equally among replicas. The SD is 10 as before. Replicas obtain LQs respectively of 21 for R_{i0} ($PR_{i0} = 0.4$) and 26 for the other three replicas ($PR_{i1} = PR_{i2} = PR_{i3} = 0.2$). The sum of all LQ yields only 99. As such, there is one message left to assign after considering the "easily partitionable" work. R_{i0} has the highest PR and is thus furthest away from a fair assignment. Hence, we increase its message assignment by 1, from 21 to 22.

5.3 Retransmissions

Two issues remain to ensure eventual delivery with stake: (1) the process of apportionment may select so few senders and receivers ($q < \mathbf{u}_s + \mathbf{u}_r + 1$) that reliable delivery is not guaranteed. (2) if the total stake across both RSMs is large, then all safe $q > \mathbf{u}_s + \mathbf{u}_r + 1$ may be too large to achieve parallelism. For example, if the total stake of RSM $\mathcal{R}_s$ is $\mathbf{\Delta}_s = 4$ and RSM $\mathcal{R}_r$ is $\mathbf{\Delta}_r = 4,000,000$ then $q > \mathbf{u}_s + \mathbf{u}_r + 1 = 1{,}333{,}335$ which is an unrealistic number of messages to generate in a time quantum.

The core issue present is that for reliable delivery, every message m_k, across all resends, must be sent and received by nodes whose stake, together, exceeds $\mathbf{u}_s + \mathbf{u}_r + 1$. This couples the number of resends needed to the (effectively unbounded) amount of stake in a network, and forces us to use increasingly large time quanta. Consider two networks with identical large stake values. If $\mathcal{R}_s$ and $\mathcal{R}_r$ both have $\mathbf{\Delta}_s = \mathbf{\Delta}_r = 4,000,000$, with each node having $1,000,000$ stake, each message send would pair replicas with 1,000,000 stake and we would reach $\mathbf{u}_s + \mathbf{u}_r + 1 = 2,666,667$ stake after 3 message sends even without apportionment. This contrasts with our original example ($\mathbf{\Delta}_s = 4$,$\mathbf{\Delta}_r = 4{,}000{,}000$). Each replica in $\mathcal{R}_s$ and $\mathcal{R}_r$ is equally trusted, but we require

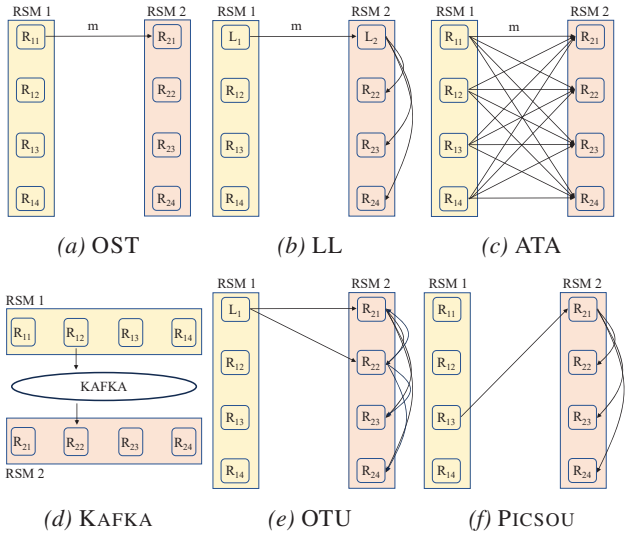


Figure 6: C3B *baseline summary.*

$\mathbf{u}_s + \mathbf{u}_r + 1 = 1{,}333{,}335$ resends solely because the *relative* value of stake in the two RSMs has changed.

Thankfully, this is not fundamental. To sidestep this issue, PICSOU proportionally scales up the weights of the two communicating RSMs to their Least Common Multiple (*LCM*), and handles failures with the scaled stake values independent of apportionment. For instance, assume that the total stake of RSM $\mathcal{R}_s$ is $\mathbf{\Delta}_s$, RSM $\mathcal{R}_r$ is $\mathbf{\Delta}_r$ and the $LCM = \text{lcm}(\mathbf{\Delta}_s, \mathbf{\Delta}_r)$. PICSOU scales the two RSMs as follows:

1. Compute the multiplicative factor ψ for each RSM: $\psi_s = \frac{LCM}{\mathbf{\Delta}_s}$ and $\psi_r = \frac{LCM}{\mathbf{\Delta}_r}$.
2. Multiply the stake of each replica with the multiplicative factor of its RSM.

Scaling up RSMs is only necessary during message failures, allowing to keep message quanta small in the common-case. A replica thus uses the scaled up RSM weights upon receiving its first duplicate quack for a message m.

6 Evaluation

PICSOU aims to offer good performance in the common-case, while remaining robust to faults when failures do arise. We aim to answer the following three questions.

1. How does PICSOU perform in the common case (§6.1)?
2. How does PICSOU remain robust to failures (§6.2)?
3. How does PICSOU perform in real applications (§6.3)?

Implementation We implemented PICSOU in $\approx$ 4500 lines of C++20 code with Google Protobuf v3.10.0 for serialization and NNG v1.5.2 for networking [2]. PICSOU is designed to be a plug-and-play library that can be easily integrated with existing RSMs. We evaluate PICSOU against five other comparable protocols (Figure 6).

1. **One-Shot (OST)**: In OST, a message is sent by a single sender to a single receiver. OST is *only* meant as a performance upper-bound. It does not satisfy C3B as message delivery cannot be guaranteed.
2. **All-To-All (ATA)**: In ATA, every replica in the sending RSM sends all messages to all receiving replicas ($O(\mathbf{n}_s \times \mathbf{n}_r)$ message complexity). Every correct receiver is guaranteed to eventually receive the message.
3. **Leader-To-Leader (LL)**: The leader of the sending RSM sends a message to the leader of the receiving RSM, who then internally broadcasts the message. This protocol does not guarantee eventual delivery when leaders are faulty.
4. **KAFKA**: Apache KAFKA is the de-facto industry-standard for exchanging data between services [56]. Producers write data to a Kafka cluster, while consumers read data from it. Kafka, internally, uses Raft [69] to reliably disseminate messages to consumers. We use Kafka 2.13-3.7.0.
5. **OTU**: GeoBFT [42, 44] breaks down an RSM into a set of sub-RSMs. Much like LL, GeoBFT's cross-RSM communication protocol, OTU, has the leader of the sender RSM send its messages to at least $\mathbf{u}_r + 1$ receiver RSM replicas. Each receiver then internally broadcast these messages. When the leader is faulty, replicas timeout and request a resend. OTU thus guarantees eventual delivery after at most $\mathbf{u}_s + 1$ resends in the worst-case (for $O(\mathbf{u}_r * \mathbf{u}_s)$ total messages).

RSMs. We consider four representative RSMs.

1. **File**: An in-memory file from which a replica can generate committed messages infinitely fast. This is a baseline to artificially saturate the C3B protocols.
2. **Raft [37]**: A widely used CFT RSM, used in services like Kubernetes Cluster. We run Etcd's Raft version v3.0.
3. **ResilientDB [45]**: A high performance implementation of *PBFT* [28], a well-known representative BFT protocol.
4. **Algorand [40]**: A popular PoS blockchain protocol [40].

Experimental Setup. We deploy up to 45 GCP c2-standard-8 nodes (Intel Cascade Lake, 8vCPU, 32 GiB RAM, 15 GBits/s). Each experiment runs for 180 seconds (30 second warmup/cool down). All experiments run PICSOU with a ϕ-list of 200k and 256 bits for 0.1kB and 1MB messages, respectively (best results for our specific network setup). We further assume that RSMs forward all messages to the other RSM, as this represents a worst-case scenario for PICSOU. As is standard [43, 68, 78, 84, 97], unless stated otherwise, replicated operations in our experiments are no-ops , which ensures that the bottleneck is not execution.

Metrics. RSM *throughput* is the number of consensus invocations completed at an RSM per second; C3B *throughput* is the number of completed C3B invocations per second. When baselines, like OST, do not acknowledge received messages, we calculate C3B throughput as the number of unique messages sent from sender RSM to receiver RSM.

6.1 File RSM Common Case Performance

Our first set of experiments aim to stress test the six C3B protocols (PICSOU, OST, ATA, LL, OTU, and Kafka) *without* failures. We use the "infinitely fast" File RSM to saturate

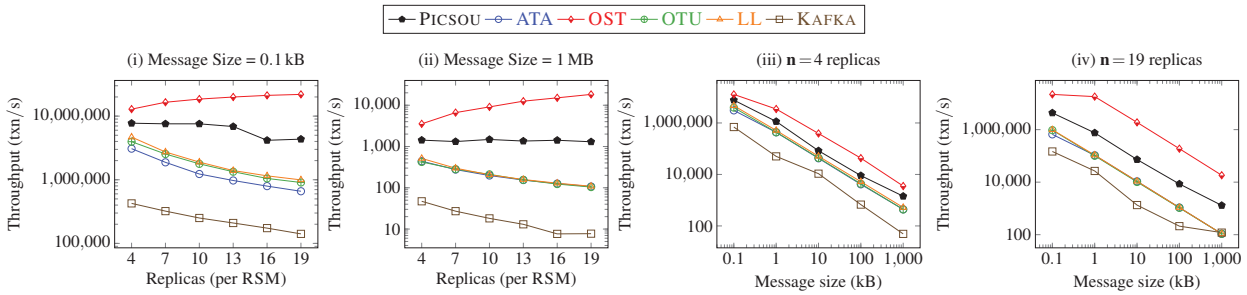


Figure 7: Throughput of C3B *protocols as a function of network size and message size*

all C3B implementations. In all cases, we include the OST line as the upper-bound of our networking implementation.

Varying number of replicas in each RSM. We first consider the relative performance of PICSOU as a function of the network size. We fix the message size to 0.1 kB and 1 MB and increase the number of replicas in each RSM from 4 to 19 (Figure 7 (i)-(ii)). For small network sizes, PICSOU outperforms ATA by a factor of 2.5× (small messages) and 3.2× (large messages) and in larger networks it increases to 6.6× and 12.1×. PICSOU sends only a linear number of messages, while ATA must send a quadratic number of messages. Like PICSOU, LL and OTU send a linear number of messages, but quickly bottleneck at the leader since it needs to send every message. OST's performance, as expected, increases with network size as increasing the number of replicas increases the number of parallel messages. Kafka performs significantly worse in all cases, as it internally runs consensus.

Varying Message Size. In Figure 7 (iii)-(iv), we fix the size of each RSM to $n=4$ (small) and $n=19$ replicas (large) and increase the message size from 0.1 kB to 1 MB. As expected, the performance of each C3B implementation drops as a linear function of the message size. Note that PICSOU performs relatively better than other protocols for large message sizes as they hide the moderate compute overheads introduced by the system. For instance, on a large network PICSOU performs over 12× better than ATA, LL, and OTU for large messages. Instead, for small messages, PICSOU only performs 6.6×, 4.4×, and 4.9× better (respectively).

Impact of Stake Next, in Figure 8 (i), we study how well PICSOU performs for weighted RSMs when stake distribution becomes unequal. We fix the message size to 100 B.

Consider 1) two RSMs where the throughput is *throttled* and one replica in each RSM gets increasingly more stake; 2) two RSMs where throughput is not throttled, but one replica still gets a larger share of stake over time. Our aim is to demonstrate that PICSOU does not lose any performance under unequal stake distributions.

We run two experiments. First, we artificially throttle the File RSM such that PICSOU cannot transmit over 1M txns/s, regardless of the stake distribution (flat 1M lines on the graph). Next, we allow each node to have access to an unmodified File RSM. In these experiments PICSOU i refers to the setting where the high-stake node has $i\times$ more stake

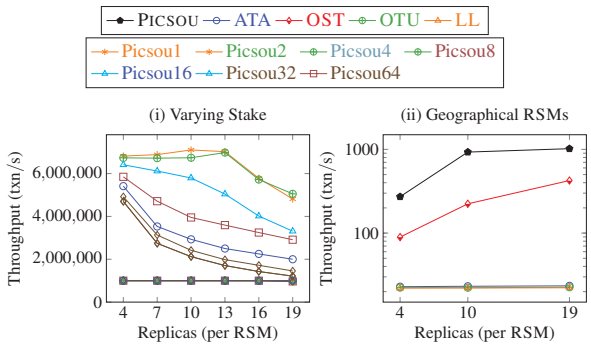


Figure 8: Impact of Stake and Geo-replication.

than other nodes. Initially, shifting the stake distribution to one node does not affect performance as the high stake node can handle the additional load. Eventually, however, this node becomes a bottleneck, thus causing throughput to decrease.

Geo-replication In Figure 8(ii), we run geo-replicated experiments by deploying one RSM in US-West and the other RSM in Hong Kong (cross-region bandwidth, pair-wise is 170 Mbits/sec, RTT 133 ms). We fix the message size to 1 MB and vary RSM size from 4 to 19. The lower bandwidth across pairs of machines disproportionally affects ATA, LL, and OTU. PICSOU outperforms ATA, by 12× (for network size 4) and 44×, (for network size 19). Somewhat counter-intuitively, the performance of both PICSOU and OST increase as a function of network size; increasing the number of receivers gives senders access to more bandwidth in Google Cloud. PICSOU intentionally has its senders send to multiple receivers and thus (artificially) outperforms OST, which fixes unique sender-receiver pairs.

6.2 Impact of failures

We now consider performance under failures.

Crash Failures. In this experiment, we crash 33% of the replicas in each RSM (Figure 9 (i)); message size set to 1 MB and ϕ-list size as 256. PICSOU's performance drops by a factor of 22.8% − 30.5%. This is expected: PICSOU, by default, fully maxes out links with "useful" information. Removing a third of the links thus removes a third of the available bandwidth. Nonetheless, PICSOU continues to outperform ATA, OTU, and LL by at least 2× on small networks, and up to 8.9× on larger networks.

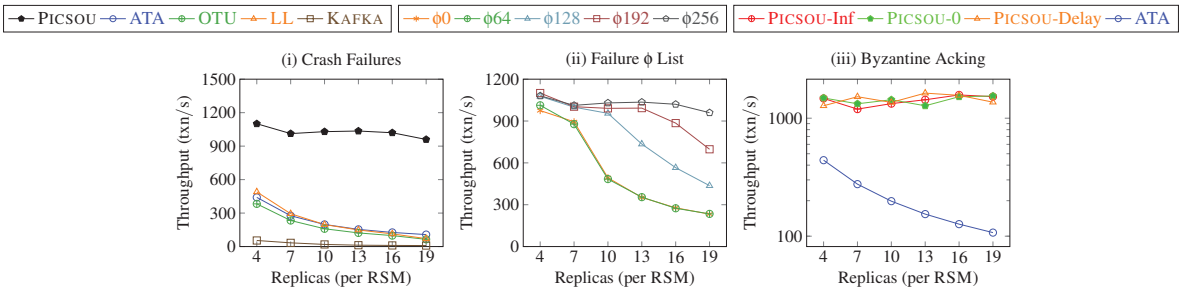


Figure 9: Effects of Failures on PICSOU.

Byzantine Failures. Next, we consider the impact of Byzantine failures in the system. While it is impossible to model all arbitrary failures, we consider four main classes of attacks. Malicious nodes can (1) send invalid, uncommitted messages, (2) collude to drop long sequences of messages $\mathbf{u}_s + \mathbf{u}_r$ times, (3) selectively drop messages, and (4) send incorrect acknowledgments. The first attack amounts to a DDOS attack (as correct replicas will discard invalid messages) and is thus out of scope. PICSOU defends against the second attack by assigning node IDs using a verified source of randomness (the probability that all byzantine nodes get assigned contiguous node IDs is negligible). We focus on the last two scenarios.

1. **Impact of ϕ-list scaling on Byzantine failures.** ϕ-lists bound the possible performance drop from malicious nodes selectively dropping messages. We again assume 33% of replicas are faulty in both RSMs (Figure 9 (ii)), this time Byzantine. We consider a message size of 1 MB. Our results illustrate that the larger ϕ-list size helps PICSOU quickly recover from Byzantine failures, despite the larger ϕ-list increasing metadata sizes. We observe that a ϕ-list size of 256 is optimal for recovering from the 33% Byzantine attacks. As the network gets larger, the time it takes to complete a full broadcast gets longer, which increases the latency to confirm a delivery. Thus, more messages can be dropped before we can detect that they are dropped, hence the larger ϕ-list.
2. **Sending incorrect acks.** Malicious nodes can choose to lie in their acknowledgments. We simulate this behavior in Figure 9 (iii) by having malicious nodes send acks for overly high sequence numbers (Picsou-Inf), overly low ones (Picsou-0) or offset by ϕ (Picsou-Delay). We find that this behavior is much less harmful than simply crashing. Correct nodes wait for a quorum of $\mathbf{u}_r + 1$ matching acks in order to consider the message delivered, and thus already assume that $\mathbf{u}$ of those acks will be lies. Lying about an ack thus only temporarily delays the formation of a quorum.

6.3 Application Case Study

We now study impact on real-world applications (Section 1).

Disaster Recovery. Disaster recovery (DR) ensures continued fault-tolerance in the presence of full datacenter outages, and is a popular feature of modern cloud environments [14, 16, 33, 41]. DR deployments often implement cross-datacenter RSM mirroring over Kafka, where the Kafka cluster is located in the receiving datacenter. We run Etcd DR [33] by deploying two Etcd RSMs in two distinct datacenters, one in GCP region us-west-4 and the other in us-east-5. Communication is unidirectional for DR, since only a single sending RSM is sending data to the mirrored RSM and the mirrored RSM does not have any information to send back (other than acks).

Etcd DR invokes PICSOU on all *put* transactions and assigns them a new, sequential, internal sequence number. This new sequence number is necessary as DR only applies to a subset of Etcd transactions (just puts, not gets or reconfiguration). The receiving RSM thus simply applies all put transactions in sequence number order.

In Figure 10 (i), we plot the throughput of Etcd DR (in MB/s) with various C3B protocols for different message sizes; each RSM has 5 replicas. OST achieves maximum theoretical throughput for an Etcd cluster running a C3B protocol; ETCD is the baseline for maximum throughput from a single Etcd RSM without any communication; one can only transmit messages as fast as Etcd commits them. There are two primary resource bottlenecks in the system: the cross-region network bandwidth and Etcd's disk goodput (since it synchronously writes each transaction it commits to disk). ATA broadcasts every message to all machines, so its throughput is bottlenecked by the cross-region network bandwidth (50 MB/s). Similarly, OTU and LL are bottlenecked because they limit the number of nodes sending unique messages over the network in parallel. In contrast, PICSOU shards the set of messages across all sending nodes, so each node uses 50 MB/s bandwidth to send $1/5$-th of the messages (5 nodes per RSM). Thus, PICSOU has an effective 250 MB/s of bandwidth available, resulting in saturating Raft's disk goodput of 70 MB/s. In case of KAFKA, we can only deploy 3 nodes, at most 3 shards, so it can achieve at most 150 MB/s. KAFKA can still can achieve potentially the same goodput as PICSOU. However, in our testing, KAFKA was still unable to achieve optimal performance given its sensitivity to high network latency.

Data Sharing and Reconciliation As described in §1 (Figure 10(ii)), there are operational and sovereignty concerns associated with managing a single RSM across trust domains. We implement the data reconciliation application described in [75]. In this setup, two distinct entities, Agency A and Agency B, run their own Etcd RSM but exchange data to ensure that any shared state remains consistent. Specifically,

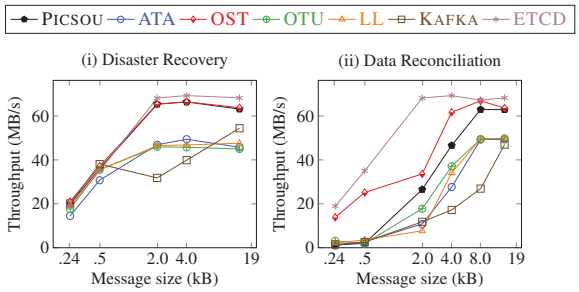


Figure 10: Disaster Recovery and Data Reconciliation.

each RSM sends *key-value updates* for shared data. The receiver then checks whether the values match and takes remedial action if not. Communication between RSMs is bidirectional. ATA, LL, OTU, and PICSOU all behave similarly to the performance discussed in the disaster recovery experiment, albeit with a lower starting goodput since there is extra processing time needed for looking up keys and comparing their values. KAFKA had unusually low performance since we were running into a known issue with high latency KAFKA consumers which are not addressed in these result. We are in the process of addressing this.

Decentralized Finance. Our final application implements a blockchain bridge, designed to foster interoperability between chains [29], for instance for asset transfer. We implement an asset transfer application across three types of wallets: (1) two POS Algorand chains, (2) two traditional permissioned PBFT ResilientDB [44, 45] chains, and (3) interoperability between ResilientDB to Algorand chains. Algorand's base throughput with another Algorand instance is 120 blocks/second. ResilientDB's base throughput when communicating with another ResilientDB cluster is ≈6000 batches/second (of size 5kB). The cross-chain throughput when Algorand sends to ResilientDB is 135 blocks/second. PICSOU has minimal impact on the throughput of any of the RSMs, with less than 15% decrease in throughput in the worst case. This decrease in throughput is independent of node stake. Latency will instead increase proportionally to network size – this property is fundamental to PICSOU's high throughput, but may be unacceptable in some large scale blockchain or RSM deployments. (2) PICSOU successfully handles throughput differences between RSMs; the slow Algorand RSM efficiently communicates with the much faster ResilientDB RSM.

7 Related Work

The problem of reliably sending messages within groups of participants through *reliable broadcast* or *group communication* is well-studied [22, 24, 26, 39, 47, 48, 62, 76], in both the CFT and BFT setting [4, 5, 23, 24, 26, 86]. These works consider communication among groups but do not consider communication between groups. PICSOU leverages the internal guarantees provided by these communication primitives to build a group-to-group communication primitive, C3B.

Logging Systems. Shared logs are a popular way for reliably exchanging messages [18, 27, 36, 52, 53, 56, 61, 88, 91]. Systems such as Kafka [56], RedPanda [73], Delos [17] have become industry standards [17]. While these systems work well in the CFT setting, they are not directly applicable to the BFT setting: this log becomes a central point of attack. Moreover, most of these systems use relatively heavyweight fault tolerance: Kafka, for instance, internally makes use of Raft.

Communication between RSMs. Two lines of work have considered communication between RSMs, but in different contexts. First, Aegean [6] makes a similar observation as this paper: it highlights that replicated services rarely operate in a vacuum and must instead frequently communicate. However, Aegan solves a strictly orthogonal problem. It focuses on how to correctly replicate services that can issue nested requests to other (possibly replicated) services. Aegean presents the design of a shim layer that exists between replicated service and backend service and manages all the communication/data storage. Second, Byzantine fault tolerant communication between RSMs has been a topic of interest in the context of *sharded* BFT systems that view each shard as an independent RSM. These shards periodically need to communicate with each other to process cross-shard transactions [9, 35, 42, 58, 71, 74, 78, 98, 100]. Most of these systems simply adopt the all-to-all communication pattern between the shards that we evaluate in §6. GeoBFT [44] and Steward [8] are two exceptions. Steward uses a hierarchical consensus architecture; all communication between the clusters is managed by a designated primary cluster, which internally replicates requests via Paxos. GeoBFT uses OTU.

Blockchain bridges. With the rise of blockchain technology and cryptocurrencies [11, 12, 13, 46, 54, 60, 64, 65, 77, 85] there is a new found interest in blockchain interoperability [20, 29, 30, 49, 55, 90, 92, 99]. These works focus on the *correct* conversion of assets from one blockchain to the other. They can be broadly clustered into two groups (1) *blockchain bridges*, and (2) *trusted operators*. A blockchain bridge requires a replica of the sending RSM to send a committed contract to a replica of the receiving RSM. Recently, several such blockchain bridges have popped up [15, 72, 82]. Unfortunately, they provide few formal guarantees, which has led to massive financial attacks and hacks [92, 96, 99]. Moreover, these bridges continue to be impractical because of their high cost [96]. Trusted operator systems are, in contrast, much more practical [57, 90, 95], but as the name suggests, they require centralized management. Works like Thema [63] instead use BFT RSMs to communicate between two non-replicated services.

8 Conclusion

This paper introduces the C3B primitive and proposes PICSOU, an efficient implementation of C3B. We show that, by borrowing techniques from TCP and adapting these to the crash and BFT context, we can develop a solution that allows RSMs to efficiently exchange messages.

References

[1] Picsou: Appendix. https://arxiv.org/abs/2312.11029.

[2] Picsou: Artifact. https://github.com/gupta-suyash/BFT-RSM.

[3] The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure. https://aptosfoundation.org/whitepaper#view-whitepaper-pdf.

[4] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected o(1) rounds, expected communication, and optimal resilience. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers*, page 320–334, Berlin, Heidelberg, 2019. Springer-Verlag.

[5] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 331–341, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Remzi Can Aksoy and Manos Kapritsos. Aegean: Replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 385–398, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Algorand Foundation. General FAQ, 2022. https://www.algorand.foundation/general-faq.

[8] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010.

[9] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. *SharPer: Sharding Permissioned Blockchains Over Network Clusters*, page 76–88. Association for Computing Machinery, New York, NY, USA, 2021.

[10] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018.

[11] Maria Apostolaki. *Routing Security of Cryptocurrencies*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2021.

[12] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant byzantine fault tolerance. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306. IEEE, 2013.

[13] Alex Auvolat, Yérom-David Bromberg, Davide Frey, and François Taïani. BASALT: A rock-solid foundation for epidemic consensus algorithms in very large, very open networks. *CoRR*, abs/2102.04063, 2021.

[14] AWS. Disaster Recovery of Workloads on AWS: Recovery in the Cloud. https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html.

[15] Axelar. Axelar network: Connecting applications with blockchain ecosystems, 2021. https://axelar.network/axelar_whitepaper.pdf.

[16] Azure. Backup and disaster recovery. https://azure.microsoft.com/en-us/solutions/backup-and-disaster-recovery.

[17] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632. USENIX Association, November 2020.

[18] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 325–340, 2013.

[19] M.L. Balinski and H.P. Young. Chapter 15 apportionment. In *Operations Research and The Public Sector*, volume 6 of *Handbooks in Operations Research and Management Science*, pages 529–560. Elsevier, 1994.

[20] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *ACM Comput. Surv.*, 54(8), oct 2021.

[21] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. Geo-distribution of actor-based services. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[22] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, jan 1987.

[23] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[24] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, oct 1985.

[25] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes: up and running*. " O'Reilly Media, Inc.", 2022.

[26] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings International Conference on Dependable Systems and Networks*, page 167, Los Alamitos, CA, USA, jun 2002. IEEE Computer Society.

[27] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[28] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[29] Chainlink. Chainlink cross-chain interoperability protocol, 2023.

[30] João Otávio Massari Chervinski, Diego Kreutz, Xiwei Xu, and Jiangshan Yu. Analyzing the performance of the inter-blockchain communication protocol. *CoRR*, abs/2303.10844, 2023.

[31] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 277–290. ACM, 2009.

[32] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 153–168. USENIX, 2009.

[33] Confluent. Geo-replication with Cluster Linking on Confluent Cloud, 2024.

[34] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[35] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, pages 123–140. ACM, 2019.

[36] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 325–338, USA, 2020. USENIX Association.

[37] etcd. Etcd raft. https://github.com/etcd-io/raft.

[38] Ethereum Foundation. Staking withdrawls, 2023.

[39] P.T. Eugster, R. Guerraoui, and P. Kouznetsov. /spl delta/-reliable broadcast: a probabilistic measure of broadcast reliabillity. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 636–643, 2004.

[40] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Google Cloud. What is a disaster recovery plan? https://cloud.google.com/learn/what-is-disaster-recovery?hl=en#what-is-a-disaster-recovery-plan.

[42] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. *Fault-Tolerant Distributed Transactions on Blockchain*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2021.

[43] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: resilient concurrent consensus for high-throughput secure transaction processing. In *37th IEEE International Conference on Data Engineering, ICDE 2021*, pages 1392–1403. IEEE, 2021.

[44] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, 2020.

[45] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020*, pages 754–764. IEEE, 2020.

[46] Suyash Gupta and Mohammad Sadoghi. Blockchain transaction processing. In *Encyclopedia of Big Data Technologies*, pages 1–11. Springer, 2019.

[47] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations (Databases, Reliability, Transactions, Agreement Protocols, Distributed Computing)*. PhD thesis, Harvard University, USA, 1985. AAI8520209.

[48] Vassos Hadzilacos and Sam Toueg. *Fault-Tolerant Broadcasts and Related Problems*, page 97–145. ACM Press/Addison-Wesley Publishing Co., USA, 1993.

[49] Maurice Herlihy. Atomic cross-chain swaps. *CoRR*, abs/1801.09515, 2018.

[50] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: a raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[51] IBM. Blockchain for supply chain solutions. https://www.ibm.com/blockchain-supply-chain.

[52] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

[53] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[54] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. Hotstuff-1: Linear consensus with one-phase speculation. *CoRR*, abs/2408.04728, 2024.

[55] Aggelos Kiayias and Dionysis Zindros. Proof-of-work sidechains. In *Financial Cryptography and Data Security*, pages 21–34, Cham, 2020. Springer International Publishing.

[56] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[57] Jae Kwon and Ethan Buchman. A network of distributed ledgers, 2016.

[58] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. Scaling nakamoto consensus to thousands of transactions per second, 2018.

[59] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. Flexible advancement in asynchronous bft consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 264–280, New York, NY, USA, 2023. Association for Computing Machinery.

[60] Yang Liu, Yuxi Zhang, Zhiyuan Lin, Zhaoguo Wang, and Xuan Wang. Simulation method for blockchain systems with a public chain. *Sensors*, 22(24):9750, 2022.

[61] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A partially ordered shared log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 357–372, Carlsbad, CA, October 2018. USENIX Association.

[62] D. Malkhi, M.K. Reiter, O. Rodeh, and Y. Sella. Efficient update diffusion in byzantine environments. In *Proceedings 20th IEEE Symposium on Reliable Distributed Systems*, pages 90–98, 2001.

[63] M.G. Merideth, Arun Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 131–140, 2005.

[64] Ines Messadi, Markus Horst Becker, Kai Bleeke, Leander Jehl, Sonia Ben Mokhtar, and Rüdiger

Kapitza. Splitbft: Improving byzantine fault tolerance safety using trusted compartments. In *Middleware '22: 23rd International Middleware Conference*, pages 56–68. ACM, 2022.

[65] Johnnatan Messias, Vabuk Pahari, Balakrishnan Chandrasekaran, Krishna P. Gummadi, and Patrick Loiseau. Dissecting bitcoin and ethereum transactions: On the lack of transaction contention and prioritization transparency in blockchains. *CoRR*, abs/2302.06962, 2023.

[66] Microsoft. Azure service fabric, 2024.

[67] Microsoft. Microsoft Azure confidential ledger, 2025.

[68] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 35–48, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 305–320. USENIX, 2014.

[70] Paxos. Blockchain infrastructure for enterprises, 2024.

[71] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. Ringbft: Resilient consensus over sharded ring topology. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022*, pages 2:298–2:311. OpenProceedings.org, 2022.

[72] RainbrowBridge. Eth - near rainbow bridge, 2020.

[73] RedPanda. The state of streaming data, 2023.

[74] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. Blockchains vs. distributed databases: Dichotomy and fusion. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1504–1517. ACM, 2021.

[75] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. Ccf: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.

[76] Fred B. Schneider, David Gries, and Richard D. Schlichting. Fault-tolerant broadcasts. *Sci. Comput. Program.*, 4(1):1–15, may 1984.

[77] Lili Su, Quanquan C. Liu, and Neha Narula. The power of random symmetry-breaking in nakamoto consensus. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 39:1–39:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[78] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[79] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Andrew S. Tanenbaum and David Wetherall. *Computer networks, 5th Edition*. Pearson, 2011.

[81] Peter Tannenbaum. *Excursions in modern mathematics*. Pearson, Upper Saddle River, NJ, 9 edition, December 2008.

[82] Poly Team. Polynetwork: An interoperability protocol for heterogeneous blockchains, 2015.

[83] The MystenLabs Team. The sui smart contracts platform. https://docs.sui.io/paper/sui.pdf.

[84] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. Quepaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 281–297, New York, NY, USA, 2023. Association for Computing Machinery.

[85] Sarah Tollman, Seo Jin Park, and John K. Ousterhout. Epaxos revisited. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 613–632. USENIX Association, 2021.

[86] Sam Toueg. Randomized byzantine agreements. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, page 163–178, New York, NY, USA, 1984. Association for Computing Machinery.

[87] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), February 2015.

[88] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389, 2017.

[89] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible Proportional-Share resource management. In *First Symposium on Operating Systems Design and Implementation (OSDI 94)*, Monterey, CA, November 1994. USENIX Association.

[90] Gang Wang, Qin Wang, and Shiping Chen. Exploring blockchains interoperability: A systematic survey. *ACM Comput. Surv.*, 55(13s), jul 2023.

[91] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, 2015.

[92] Xuechao Wang, Peiyao Sheng, Sreeram Kannan, Kartik Nayak, and Pramod Viswanath. Trustboost: Boosting trust among interoperable blockchains. *CoRR*, abs/2210.11571, 2022.

[93] E. Weldon. An improved selective-repeat arq strategy. *IEEE Transactions on Communications*, 30(3):480–486, 1982.

[94] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2015.

[95] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework, 2016.

[96] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 3003–3017. ACM, 2022.

[97] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019.

[98] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948. ACM, 2018.

[99] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. Sok: Communication across distributed ledgers. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 3–36, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[100] Yuanzhe Zhang, Shirui Pan, and Jiangshan Yu. Txallo: Dynamic transaction allocation in sharded blockchain systems. In *39th IEEE International Conference on Data Engineering, ICDE*, pages 721–733. IEEE, 2023.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

FineMem: Breaking the Allocation Overhead vs. Memory Waste Dilemma in Fine-Grained Disaggregated Memory Management

Xiaoyang Wang[1], Yongkun Li[1]*, Kan Wu[2], Wenzhe Zhu[1], Yuqi Li[1], Yinlong Xu[1,3]

[1]*University of Science and Technology of China* [2]*Google*
[3]*Anhui Provincial Key Laboratory of High Performance Computing, USTC*

Abstract

RDMA-enabled memory disaggregation has emerged as an attractive approach to reducing memory costs in modern data centers. While RDMA enables efficient remote read/write operations, it presents challenges in remote memory (de)allocation. Consequently, existing systems adopt coarse-grained allocations (in GBs), leading to memory waste.

We introduce FineMem, an RDMA-connected remote memory management system that enables high-performance, fine-grained memory allocation. FineMem addresses latency and scalability challenges related to fine-grained allocations. It removes RDMA memory region (MR) registration costs from allocation paths through per-compute node MR pre-registration, while ensuring remote memory isolation using RDMA memory windows and a trusted allocation service on each compute node. It employs a lock-free, one-sided RDMA-based protocol to allocate memory chunks (e.g., 4KB, 2MB) without involving the memory node's CPU and maintains metadata consistency during compute node failures via logging. We show that FineMem reduces remote memory allocation latency by as much as 95% compared to state-of-the-art remote memory management systems. It enables memory malloc systems, key-value stores systems, and swap systems running on FineMem to achieve low memory waste with minimal overhead.

1 Introduction

The growing need for hyperscalers to reduce memory costs [18, 38, 43] has made memory disaggregation an attractive architectural approach in system design. Unlike traditional architectures that tightly couple memory with compute resources – often leading to memory underutilization [38], disaggregated memory (DM) decouples memory (memory nodes) from compute (compute nodes), and enables dynamic allocation of memory across multiple compute nodes. Over the past decade, memory disaggregation has emerged as a pivotal area of systems research [8, 22, 34, 38, 61, 62, 64, 73].

Advancements in high-speed interconnect technologies, particularly Remote Direct Memory Access (RDMA) [51], have made it feasible to implement DM systems effectively. RDMA, facilitated by RDMA-capable network interfaces (RNICs [26, 46]), allows compute nodes to perform direct memory read/write operations on remote memory nodes using one-sided operations, bypassing the memory node's CPU entirely. Minimizing CPU involvement on memory nodes is critical for achieving low latency and high throughput, especially given their limited processing power [41, 58, 65, 73].

Despite RDMA's strengths in data access, it presents significant challenges in efficient memory allocation and deallocation. Allocating memory from a memory node involves several operations: the OS in memory node needs to pre-fault physical memory pages using mmap(), registers and copies page table entries to the memory transaction table (MTT) in the RNIC, and generates on-device memory management units with capabilities, named memory regions (MRs [4]) with rkey [53]. These operations are notably time-consuming and must run on memory node CPUs — for instance, registering a 4MB memory region can take more than 480 µs.

To work around RDMA's inefficiencies in memory (de)allocation, existing disaggregated memory applications/systems (e.g., memory malloc systems [15, 59, 70], key-value store systems [41, 58, 73], and swap systems [3, 22, 48]) resort to coarse allocation strategies, often grabbing chunks of 1GB or more at a time. While this approach amortizes the overhead of allocations, it comes at a steep cost. Large memory chunks lead to significant memory underutilization because unused portions of these coarse-grained chunks cannot be reclaimed or shared safely across multiple DM systems in the disaggregated memory pool. This leaves RDMA-based DM systems facing a difficult dilemma: either suffer from crippling allocation overheads or tolerate memory waste.

In this paper, we introduce FineMem, a fine-grained remote memory management system. It delivers fine-grained memory allocation and deallocation (e.g., 4KB, 2MB, ...)

*Yongkun Li is the corresponding author.

with minimal memory waste and high performance, employing one-sided RDMA. FineMem supports flexible allocation granularities across diverse DM systems within a shared disaggregated memory pool. By using one-sided RDMA, FineMem bypasses the memory node's CPU during allocations, ensuring that it does not suffer from a scalability bottleneck [41, 58, 65, 73]. FineMem tackles three key challenges: 1) removing MR registration latency from the DM system's allocation path, 2) enabling efficient and highly concurrent allocations of free remote memory chunks, and 3) ensuring metadata consistency when facing compute node failures.

To remove MR registration overheads during allocation, FineMem builds upon a strawman approach while addressing critical isolation challenges. The strawman method pre-registers the entire remote memory space as a single MR when a memory node boots. When a DM system starts on the compute node, the memory node shares the MR rkey with the system, allowing it to allocate or access chunks from the pre-registered MR. While this strawman approach removes MR registration overhead from systems, it introduces significant isolation issues. A DM application/system can potentially access any remote memory belonging to other applications/systems. Additionally, coordinating (de)allocation across DM systems requires sharing the memory management metadata region (which contains information on free/used memory chunks), allowing any DM system to read/modify this metadata. This lack of protection poses significant security risks to the entire memory pool.

FineMem first uses the memory window (MW [14]) feature in RDMA to ensure memory access isolation across DM systems. The memory node pre-binds a MW to each chunk with a specific rkey. When (de)allocating fine-grained chunks, FineMem uses one-sided operations to acquire/invalidate rkeys, thus removing bind-related RPC at the allocation critical path. Secondly, FineMem employs compute node allocation services to protect remote memory management metadata (e.g., index, redo-log, and rkey). DM Systems use inter-process communication (IPC) to interact with the allocation service for (de)allocation, and only the allocation service holds the rkey necessary to read or modify the metadata region.

To enable efficient, highly concurrent allocations of remote memory chunks across compute nodes, FineMem employs a carefully designed two-layer bitmap tree to accelerate allocation. It effectively limits the number of RDMA round-trips to locate a free chunk. Additionally, FineMem embeds contention control information into the bitmap, thereby avoiding further contention and reducing the number of retries for allocation's RDMA Compare-and-Swap (CAS) update, which results in more predictable latency.

Finally, to ensure metadata crash consistency in the event of compute node failures, FineMem executes allocation in two steps: 1) a commit point to confirm allocation success, combined with temporary redo-log information, and 2) flushing logs and updating related bitmap while detecting inconsistencies using timestamps and commit point metadata. This mechanism ensures metadata consistency for the search index and redo-log, thereby enabling reliable failure recovery.

We have implemented FineMem on Linux and demonstrate its portability across various systems, including memory malloc systems (such as jemalloc [19], mimalloc [36]), key-value store systems [41, 58, 73], and kernel swap systems [2, 3, 22, 64]. Based on mimalloc [36], we built FineMem-User, a user-space DM object malloc system that uses FineMem's DM allocation APIs. We also ported a popular swap system, FastSwap [3], and a DM-native key-value store [58] to integrate with FineMem (Sec. 5). Our evaluations show that FineMem reduces the latency for remote memory allocation by up to 95% compared to state-of-the-art designs. For memory systems, FineMem improves memory utilization by 2.25× to 2.8× compared to coarse-grained memory management, while introducing minimal overheads of only 2.5%-4.1%.

In the rest of this paper, we demonstrate existing memory management designs for DM and their limitations (Sec. 2); discuss FineMem's design (Sec. 3, 4, 5); characterize FineMem's performance through allocator benchmarks and end-to-end applications on memory systems (Sec. 6); discuss the limitations and potential extensions of FineMem (Sec. 7); review related work (Sec. 8); and conclude (Sec. 9). The implementation of FineMem is available at https://github.com/ADSLMemoryDisaggregation/FineMem.

2 Background and Motivation

2.1 Remote Memory Management and RDMA

Memory disaggregation introduces a remote memory pool shared by compute nodes. In this work, we focus on RDMA-connected DM, where compute nodes access shared memory nodes via RDMA primitives.

RDMA supports two communication patterns: Remote Procedure Call (RPC) using send/receive operations, and direct remote memory access through one-sided read/write operations that bypass the receiver cores. One-sided operations require external memory authorities on RDMA NICs. Before clients can perform one-sided operations, a memory region (MR) must be registered [4] with the RNIC. During registration, the driver pins the physical memory pages and maintains page mappings in an on-chip page table [15]. The registration process returns a region key (rkey) that allows clients to directly access the MR. Clients within the same Protected Domain (PD) and possessing the appropriate rkey can freely read/write the contents of the MR. Consequently, malicious clients possessing the correct rkey could launch attacks by modifying the MR contents. Similar to MR, the RDMA specification defines memory windows (MWs) as an external memory protection mechanism built on top of MRs, providing rkeys that grant fine-grained access capabilities to contiguous ranges of memory.

State-of-the-art DM usage typically relies on disaggregated memory systems, which can be categorized into two types: DM-transparent and DM-native systems. DM-transparent systems leverage the kernel's swap mechanism, enabling unmodified applications to run on a disaggregated memory pool, thereby transparently extending local memory [3, 10, 22, 23, 57, 64]. In contrast, DM-native systems introduce new programming models [15, 54, 59, 62, 70] (e.g., memory malloc systems that allow applications to operate on a memory pool through APIs for remote memory allocation and access) or algorithmic redesigns [41, 58, 73] (e.g., kv-store systems) to fully exploit the performance of DM. Recent researches [10, 23, 55] have also explored hybrid solutions that combine elements of both approaches. For instance, FDP-DC [55] supports both DM-transparent and DM-native applications/systems within a single memory pool. Consequently, DM pools now support a variety of DM applications and systems running concurrently, as discussed in [10, 15, 23, 55, 59, 62, 70]. To accommodate such diversity, the DM pool must support sharing among multiple types of DM applications/systems.

For all these DM systems, DM pools require memory managers to allocate and free memory based on runtime demands. Similar to memory management in single-node operating systems, DM systems interact with the DM memory manager to allocate and deallocate memory chunks (e.g., 4KB, 2MB). However, unlike memory management in single-node OSes, RDMA-based memory (de)allocation is expensive.

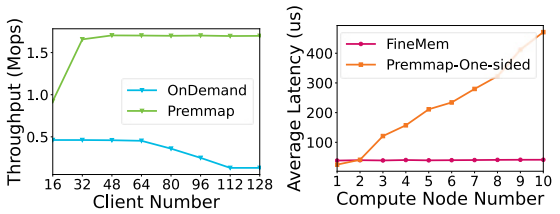
2.2 Remote Memory Allocation Challenges with RDMA

Memory allocation over RDMA differs from local memory allocation in two significant ways: the need for MR registration and the (unsurprising) fact that allocations occur over the network. In this section, we quantitatively evaluate how these two factors affect application performance.

Memory Region Registration is Costly. The overhead associated with RDMA MR registration is a well-documented problem [15, 24, 34, 45, 56, 62]. MR registration, which involves pinning physical memory pages and updating the RNIC's page tables, incurs significant overhead (e.g., a 4MB remote memory registration takes 480 µs).

In Fig. 1(a), we illustrate the impact of MR registration costs on end-to-end application performance using a DM kv-store system, FUSEE [58]. We compare on-demand MR registration for each 4KB chunk allocation with a baseline using a pre-registered large memory region (Premmap, no MR registration during runtime). In a YCSB-A workload with a 50:50 search-to-update ratio, the on-demand approach achieves only 26.7% of the throughput of Premmap (64 clients), highlighting that frequent MR registration during runtime is impractical for remote memory allocations.

Frequent Remote Allocations Over Network are Costly.



(a) MR registration impact to kv-store.

(b) Chunks allocation latency of one-sided approach.

Figure 1: Remote Memory Allocation Challenges with RDMA: (a) MR registration overheads. (b) Scalability issues in existing one-sided RDMA-based approach (due to metadata design).

Beyond MR registration, handling remote allocations presents challenges because the operations occur over the network, introducing additional latency and potential bottlenecks.

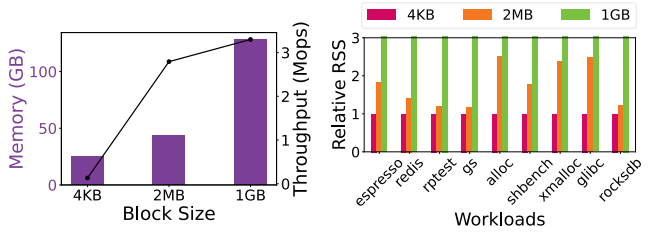
Systems that delegate memory allocation to memory nodes via RPC (e.g., FUSEE [58]) face scalability issues, as memory nodes often have limited processing power. Memory allocation involves traversing metadata, and frequent (de)allocations impose significant processing demands. Similar to prior works [41, 65, 73], we found that the RPC-based approach becomes bottlenecked by the memory node's compute power as more clients are added. For example, in Fig. 1(a), FUSEE's throughput (even with Premmap) plateaus after 32 clients. FUSEE OnDemand encounters additional scalability issues when the number of clients exceeds 64. This highlights the need for a one-sided RDMA-based allocation approach, as used by systems like CXL-SHM [70] and our proposed FineMem.

While avoiding bottlenecks at memory nodes, existing systems using one-sided access for allocations still experience unpredictable network round-trips due to their metadata designs. In Fig. 1(b), we evaluated a general allocation benchmark from ThreadTest [6], where multiple threads repeatedly allocate and free chunks from the memory node. To isolate the effects of MR registration costs, we pre-registered sufficient memory. As shown, the latency of existing one-sided approaches (e.g., Premmap-One-sided built on CXL-SHM [70]) deteriorates significantly as the number of clients (32 threads each) increases. This degradation is caused by a high number of one-sided RDMA Compare-and-Swap (CAS) retries over the network for each allocation, which occur when there are many concurrent allocations (details in Sec. 3.1).

In summary, remote memory allocation and deallocation over RDMA face both the MR registration cost challenge and scalability issues due to network overheads and the limited processing power of memory nodes.

2.3 The Allocation Overhead vs. Memory Waste Dilemma in Existing Works

Due to the challenges with MR registration and other operations, existing systems either rely on statically pre-mmapped



(a) Trade-off on block sizes, using OnDemand DM allocation.

(b) Applications' Resident Set Size (RSS) using mimalloc with huegpages.

Figure 2: Dilemma in KV-Store and General Workloads: (a) Kv-store reaches better performance with memory wastes. (b) Memory fragmentation with hugepage granularities (1GB's RSS is collapsed to no more than 3).

regions [27, 54, 54, 58, 70, 73] or allocate memory from the remote memory pool at a coarse granularity (e.g., GBs) [2, 3, 8, 15, 22, 48, 59, 64, 68]. However, these approaches can introduce significant memory waste. Next, we characterize the tradeoff between allocation overhead and memory waste using kv-store and general workloads.

KV-Store. Using the YCSB-A workload with all key-value pairs reside in remote memory, we evaluated FUSEE's performance and remote memory usage on different block sizes (4KB, 2MB, and 1GB), as shown in Fig. 2(a). In FUSEE, updates are performed out-of-place, triggering memory allocation and deallocation, as well as memory fragmentation.

The results demonstrate a clear tradeoff between allocation overhead and memory waste. For instance, 2MB allocation granularity significantly reduces memory fragmentation (hence memory usage) compared to 1GB, which is a key optimization employed by FUSEE. However, even with FUSEE's optimizations, reducing the granularity from 1GB to 2MB results in a nearly 17% drop in kv-store throughput. This performance decline is attributed to the increased frequency of allocations, each incurring significant overhead and impacting end-to-end performance. Similarly, 4KB allocation granularity further reduces memory fragmentation but exacerbates the tradeoff due to even higher allocation overhead.

General Workloads. Fragmentation caused by different allocation granularities is a well-studied issue for general workloads, too. For instance, numerous single-node memory allocator studies have shown that larger page sizes (e.g., 2MB or 1GB) typically result in higher memory fragmentation and waste [36, 47, 52, 59, 60, 72].

In Fig. 2 (b), we used mimalloc to evaluate several benchmarks and applications, illustrating this behavior. The results show that fine-grained allocations (e.g., 2MB) significantly reduce memory fragmentation compared to larger granularities like 1GB. For specific workloads, such as shbench, extremely fine-grained allocations like 4KB are particularly effective in minimizing memory waste.

Summary. Existing RDMA-based DM systems face a tough choice: either endure significant allocation overheads or accept substantial memory waste.

In this paper, we tackle the question: Can fine-grained memory allocation be achieved with minimal overheads? Successfully addressing this challenge would reduce memory waste in disaggregated memory pools, leading to lower memory costs in modern data centers. It would also allow systems under high load to run more tasks concurrently, thereby improving overall throughput, as we will demonstrate in swap system evaluations (Sec. 6.3). Additionally, fine-grained memory allocation simplifies porting emerging applications to disaggregated memory setups. For instance, applications' local memory allocators (e.g., jemalloc, tcmalloc) could seamlessly integrate into disaggregated memory systems by replacing local mmap() calls (typically allocating memory in KB/MB units) with remote memory allocations (Sec. 6.3).

Finally, we focus on scenarios where a growing class of DM systems share a common pool of remote memory [55], as discussed in Sec. 2.1. These systems span a variety of applications, such as key-value stores (e.g., FUSEE [58]), indexing services (e.g., SMART [41]), AI inference workloads (e.g., Mooncake [49]), memory malloc systems with modified applications (e.g., CXL-SHM [70]), and swap systems with unmodified applications (e.g., Fastswap [3]). This diversity creates significant challenges in maintaining proper isolation across heterogeneous systems, particularly when third-party frameworks are involved. Additionally, conducting exhaustive tests across all possible system combinations to prevent erroneous memory access is impractical. Therefore, our approach also aims to provide low-level, fine-grained memory management that ensures coordination and isolation among DM applications/systems.

Some DM memory management works have explored on-chip authorization mechanisms for fine-grained memory allocation. For example, MIND [34] offloads memory management to on-path programmable switches [7] and introduces a VMA-based capability model for different processes. In contrast, our goal is to develop a flexible, software-based memory management solution.

3 FineMem Overview

3.1 Design Elements

FineMem is a distributed remote memory management system designed to support fine-grained, high-performance memory (de)allocation in RDMA-connected DM. Its design is driven by three key elements: 1) removing MR registration latency from the allocation path; 2) enabling efficient, scalable remote memory chunk allocation; and 3) ensuring metadata consistency during compute node failures.

Removing MR Registration While Overcoming Critical Isolation Challenges. A straightforward approach to mitigate the overhead of MR registration is to pre-register the entire remote memory as a single accessible MR for

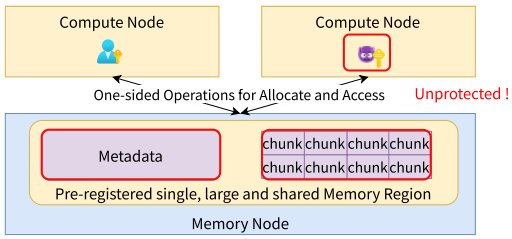


Figure 3: Isolation/Protection Challenges without MR: attacker can easily access other users' chunk and metadata.

each compute node. In this configuration, each DM system on a compute node can (de)allocate fine-grained memory chunks (subsections of the MR) by directly modifying shared allocation metadata (e.g., which chunks are free or allocated). This approach eliminates the need for runtime MR registration.

However, unsurprisingly, sharing a single MR introduces significant isolation challenges. Since the MR is associated with a single rkey, any system holding this rkey can access the entire MR, as shown in Fig. 3. This means that any entity can potentially read/write to any other user's memory chunks, leading to *privacy and security risks among systems*. Additionally, *the metadata used for memory allocation is unprotected*; each system must modify it to allocate memory, and malicious or faulty systems could corrupt or interfere with the metadata, affecting other systems.

FineMem addresses the privacy challenges by using the hardware feature of RDMA—Memory Windows (MWs) [14]. Memory windows (MWs) allow for the rapid generation and invalidation of fine-grained rkeys at a rate of one MW per microsecond. The memory node pre-binds a MW to each chunk with a specific rkey and asynchronously regenerates a new rkey with a background thread. When (de)allocating fine-grained chunks, FineMem uses one-sided operations to acquire and invalidate rkeys, thus removing MW-bind-related RPCs from the critical allocation path. This approach provides isolation between systems, as each system can only access the MWs for which it holds the corresponding rkeys. Unlike previous work that binds MWs at runtime with RPC in a single application, such as Patronus [68], FineMem focuses on one-sided MW usage for isolating multiple DM systems.

To protect the memory allocation metadata from misuse, FineMem introduces a per-compute-node runtime service (inspired by software virtualization techniques [16, 30]). This trusted service delegates remote memory allocation requests on behalf of processes, ensuring that the memory metadata is protected from DM systems' direct access.

Enabling Efficient, Highly Scalable Free Remote Chunk Allocation. As discussed in Sec. 2.2, FineMem uses one-sided RDMA access for remote allocations to prevent the memory node CPUs from becoming bottlenecks. However, this one-sided approach requires a carefully designed metadata structure to minimize network round-trips when allocating chunks.

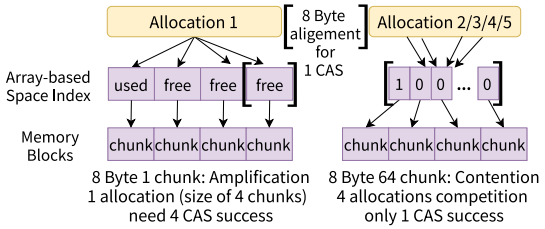


Figure 4: One-sided RDMA Based Chunk Allocation Challenge. Left: a simple chunk array may result in network amplification. Right: using bitmaps, retry time arises when multiple processes contend for the same 8B bitmap entry.

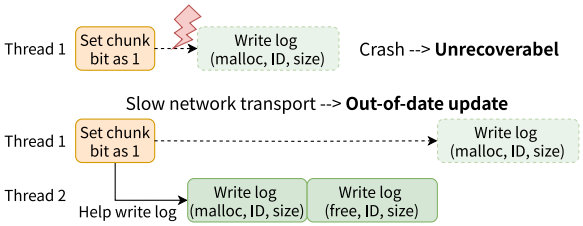


Figure 5: Two Examples of Crash Inconsistency: log lost during crash failure, and outdated update in a fail-slow failure.

Linked lists with pointer-chasing overhead. Traditional memory allocators, such as Mimalloc, XMalloc, and SFMalloc, often rely on linked-list-based free lists. However, pointer chasing—traversing pointers to locate memory blocks—introduces significant overhead when applied to remote memory, a well-known issue highlighted by prior works [41, 65, 73]. To avoid this, disaggregated memory systems often adopt array-based metadata structures, which, however, present their own challenges.

Array-based structures with search overhead. To give an example, managing free chunks using an uncompacted free-chunk array—such as in CXL-SHM [70], shown in Fig. 4 (left)—may require checking every chunk in the address space, with each check involving a round trip. This results in unpredictable and inefficient allocation times, particularly for large allocations that span multiple chunks. Alternatively, compact bitmaps (e.g., representing 64 chunks per 8 bytes) can reduce the number of round trips by enabling simultaneous search of more chunks. However, this approach still suffers from frequent Compare-and-Swap (CAS) retries, as multiple allocators may concurrently target the same 8-byte bitmap for allocation. In such cases, each allocator performs lock-free operations like CAS to allocate chunks. Yet, only one allocator can succeed, while the others must retry—each retry requiring an additional round trip (Fig. 4, right).

To address this, FineMem employs a two-layer bitmap tree structure, as shown in Fig. 7, combining the efficiency of compact bitmaps with an additional first layer to accelerate large size allocations and reduce retry attempts. Allocators read the first-layer bitmaps to allocate large memory sizes ($\geq$128KB, aligned to a power of 2) with a single CAS, or identify which *chunk groups* are empty, full, or experiencing contention (in-

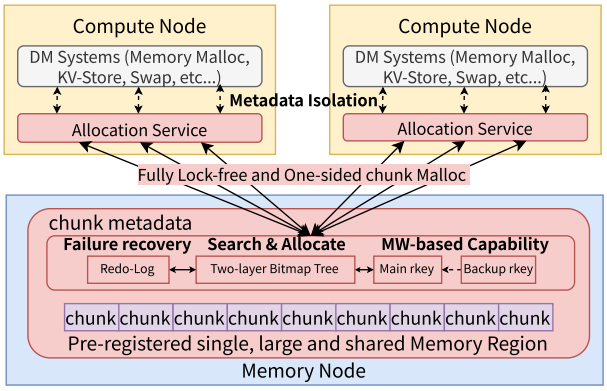


Figure 6: Remote Memory Management with FineMem.

dicating multiple concurrent allocation attempts) by a 2-bit-status in the first-layer bitmap. They then retrieve the second-layer bitmap for the selected chunk group—typically one that is neither full nor contended—to allocate small memory sizes (4KB - 64KB) with free chunks. If an allocator detects high contention on a bitmap, it can redirect its allocations to a less-contended group (Sec. 4.1).

Maintaining Metadata Consistency During Compute Node Failures. Similar to coarse-grained allocation systems, FineMem must track which memory chunks are allocated to which users to prevent memory leaks on compute node failures. To accomplish this, FineMem uses redo logging on memory nodes, written by compute nodes via one-sided RDMA, ensuring that allocations are correctly recorded. However, achieving crash consistency in a shared, bitmap-based allocation system poses unique challenges. A crash occurring before writing the redo-log can cause confusion between crash-induced and normal allocation bits. For example, as shown in Fig. 5, an inconsistency during a crash failure might cause a chunk to become unrecoverable (because it has just been marked as allocated in the bitmap but lost the detailed allocation information). Even in scenarios where other threads can assist the failed thread in writing the log, fail-slow failures can still occur, where the failed thread ultimately writes an outdated log that other threads have already written.

To address this, FineMem carefully integrates a compact temporary log into the limited 64-bit bitmap, allowing each allocation success to be temporarily logged. Each successful allocation is marked with the user's ID in the temporary log, ensuring that, in the event of a crash, the chunk can be distinguish between transient and committed states. Any thread can update (flush) this temporary log into a persistent, full log. FineMem defines a timestamp in the temporary log to prevent consistency issues due to outdated entries (Sec. 4.2). Additionally, FineMem resolves consistency issues between the two-layer bitmaps by checking both the status and temporary log consistency based on the second-layer bitmap, which acts as the commit point, ensuring the correctness of the allocation.

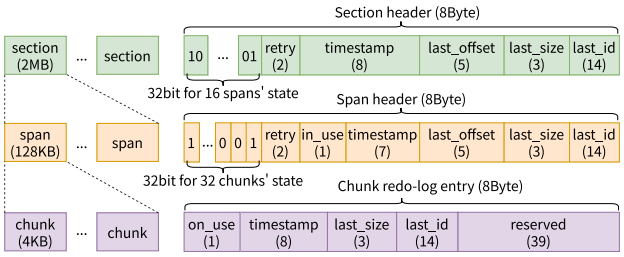


Figure 7: FineMem's Two-layer Bitmap Tree Structure with fine-grained allocation and contention control.

3.2 System Architecture

As shown in Fig. 6, FineMem integrates components from both memory nodes and compute nodes.

On the memory node, FineMem builds on core data structures that track available memory, support scalable allocations (the two-layer bitmap), manage capabilities (the MW-based capability table), and log allocations (the temporary log and full log). On the compute node, FineMem runs an allocation service on each node. It pre-mmaps the entire remote memory pool at boot time, and handles (de)allocation requests from DM systems for remote memory chunks. Once a DM system allocates a remote memory chunk, it can use the chunk for swap pages, fine-grained objects, key-value pairs, etc.

FineMem provides simple APIs, including malloc(size) and free(addr). When a DM system invokes these APIs, per-core allocation service worker threads are woken up to handle the protected remote allocations. Upon completion, the allocation service worker thread returns a DM address with an rkey to the caller. FineMem can be ported to systems like memory malloc systems [36, 63], kernel swap systems [3], key-value stores [58], etc. FineMem supports flexible allocation granularities, starting from 4KB, and ensures memory is aligned to 4KB. Systems sharing the same FineMem-managed DM pool can use different allocation granularities (4KB blocks with number of power of 2).

4 FineMem Design

In this section, we detail FineMem's design for its metadata structures for concurrent, scalable allocation, logging for crash consistency, and per-compute node allocation service.

4.1 Two-Layer Bitmaps For Concurrent, Scalable Chunk Allocation

FineMem addresses two primary remote chunk allocation challenges: 1) minimizing network round-trips during chunk searches, and 2) reducing allocation retries due to contention.

1) Two-Layer Bitmap Tree for Efficient Chunk Allocation. FineMem uses a bitmap-based free chunk index and manages the remote memory address space in two layers: sections and spans. As shown in Fig. 7, FineMem maintains

bitmaps (and metadata) for both layers. It uses a 64-bit header for each of these metadata items to align with the RDMA Compare-and-Swap (CAS) operation's work size requirement [5].

A section consists of 16 contiguous spans, with each span representing 128KB. The 32-bit bitmap for each section tracks fullness and contention (i.e., the level of concurrent allocations) of its spans, based on the most recent span allocation's CAS retry times and bitmap status, and using two bits per span to indicate its status. Span statuses 00 and 11 represent where the span is entirely free or full. Span statuses 01 and 10 represent "in-use" states, where the span is partially allocated as smaller chunks. Furthermore, 01 indicates that the span is in normal use, while 10 denotes that the span is contended.

Each span comprises 32 contiguous chunks, and a 32-bit free map to track the allocation status of these chunks. In addition to a 32-bit free map of section/span headers, FineMem reserves the remaining 32 bits for contention control and logging. FineMem uses 2 bits to track the retry count relative to two thresholds (low and high), enabling contention detection. The span header also includes an 1-bit in_use flag, which indicates whether a span is partially used or completely freed. The rest bits for logging will be discussed later.

For different allocation sizes, FineMem accesses search metadata in (up to) three steps. It first searches the section headers to identify available spans. If the requested allocation size exceeds 128KB, FineMem directly modifies the set spans to 11 using a single CAS operation. Next, it reads the span header, marks the bits corresponding to the free chunks, and updates the span header with another CAS operation. Deallocation follows a similar process with one CAS to update the related bitmap. Allocations larger than one section require multiple CAS operations to modify contiguous sections.

To optimize performance, FineMem caches section and span metadata on each compute node to reduce access latency, and batches multiple bitmap reads to amortize round-trip costs. Specifically, FineMem employs a metadata cache on each compute node, storing information for up to 64 sections and spans (a total of 512 bytes). This caching mechanism accelerates allocation requests by minimizing the need for repeated remote metadata access. When a CAS operation fails, the cache is updated accordingly. Furthermore, when cached bitmaps are exhausted or high contention is detected, the system proactively fetches new metadata blocks.

2) Contention Control to Reduce Retries. FineMem implements a simple yet effective back-pressure policy to limit contention across allocations. Each allocation header tracks the number of consecutive allocation failures. If the failure count exceeds the higher threshold (e.g., 10), the section/span is marked as contended in the header's retry bits (and also section header bitmap's span states).

Contention is detected by the allocations themselves. After a successful allocation, if the allocation process finds that the CAS failure count exceeds the threshold, it marks the section/span as 'contended (10)' to signal contention. The process will then select non-contended sections/spans, in the priority of normal (01) > empty (00) > contended (10), for future allocations.

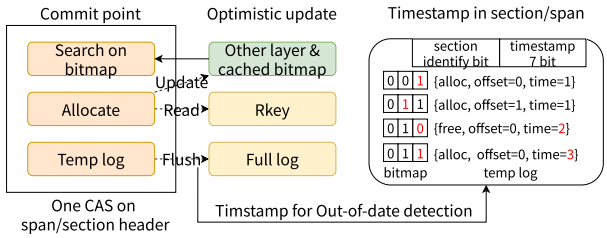


Figure 8: FineMem's Crash Consistency Design.

If a process using a contended (10) section/span detects that the failure count drops below the lower threshold (e.g., 3), it resets the status to normal (01).

In addition to the bitmap tree, FineMem maintains a per-chunk capability table and a per-chunk redo-log. Both are 8-bit aligned items in preserved linear arrays. One chunk's redo-log records information about its last allocation/free, including the allocation ID and timestamp, to facilitate recovery in case of compute node failures. The capability table stores per-chunk RDMA rkeys for access isolation.

4.2 Compute-Node-Failure Crash Consistency

FineMem implements a lock-free crash consistency mechanism by marking the CAS success to the bitmap as the commit point, complemented by a temporary log to ensure correct crash recovery. After the commit point, the system optimistically flushes the temporary log to a persistent full log with timestamp validation, and updates the other layer bitmaps, all while avoiding CAS retries and additional consistency checks to minimize overhead in maintaining consistency (Fig. 8).

1) A Compacted Commit Point. FineMem defines the commit point as the moment when an allocation successfully updates the allocation status of spans/chunks in the bitmaps (from free to used, or vice versa) through a CAS operation. This commit point, along with a temporary redo-log and timestamp, is compacted into a single CAS operation on the 64-bit section/span header.

Given the limited space available in the 64-bit section/span header, FineMem splits the redo-log into two parts: a temporary redo-log stored inline with the header's CAS requests, and a full redo-log. The temporary redo-log contains minimal information about the most recent (de)allocation of any chunks within the span/section. Specifically, it includes the last allocated chunk offset (5 bits, which allows determining whether the last request was a malloc or free by checking the bitmap), size (3 bits, sufficient given the power-of-2 size alignment), timestamp (7 bits), and user ID (14 bits).

To manage the sequence of allocations and detect out-of-date log flushing, FineMem includes a timestamp in the redo log. With a 7-bit timestamp, the value overflows after reach-

ing 127. As a result, FineMem can only determine the relative ordering of two log entries within a window of 64 timestamp updates, which corresponds to approximately 1 ms, assuming a 20 µs allocation interval. To support longer out-of-date detection, FineMem slows down the timestamp increment process. Since different chunks update their own redo-log entries, back-to-back allocations within a single section/span can share the same timestamp. Thus, FineMem's timestamp only increments when the previous operation was an allocation and the current operation is a deallocation (or vice versa), as shown in Fig .8.

2) Metadata Inconsistency Detection and Recovery. After a successful commit point, the allocation thread—or any other thread attempting to access the same section or span—is required to flush its temporary redo-log into the chunk's full redo-log before proceeding with the next operation. The system compares the timestamp of the temporary redo-log with that of the full log: if the temporary log is newer, it is flushed; otherwise, it is discarded. If the thread detects that the temporary and full logs belong to different layers (i.e., span vs. section), it further verifies the span's in_use bit to determine the most recent state and avoid acting on stale metadata.

FineMem employs lightweight, optimistic one-sided RDMA writes—instead of relying on CAS success—to update the section header's bitmap following a span commit point. During allocation, a process retrieves span headers as the commit points and checks for inconsistencies between the span and section layers. In particular, the span header's in_use bit can be used to correct an inaccurate section-layer bitmap, ensuring metadata consistency across layers.

FineMem leverages the consistency of its redo-log to enable non-blocking crash recovery for rebuilding memory allocation status. The recovery process starts by scanning the section and span metadata to flush the temporary redo-log. It then reclaims allocated chunks from the crashed node by scanning the full redo-log and regenerates rkeys for the freed chunks. To detect crashes, FineMem employs a user-space monitor on the memory node, which sends heartbeat signals every second to check the availability of each compute node.

To summarize, FineMem is designed to handle allocation/deallocation in a way that ensures crash consistency, using lock-free, one-sided RDMA verbs.

4.3 Allocation Service For Isolation/Protection

FineMem leverages memory window-based capabilities to isolate memory accesses and utilizes a per-compute node service layer to protect allocation metadata. FineMem tackles three key challenges: 1) eliminating CPU involvement on the memory node during memory window binding for allocations, 2) enabling efficient communication between user systems and the allocation service layer on the compute node, and 3) ensuring robust enforcement of isolation and protection mechanisms, even in adversarial conditions.

Table 1: MW and MR Registration Overhead Comparison.

Average Latency per 4MB (µs)	MW	MR
1*100GB Generation	1.33	456.1
25K*4MB Generation	1.34	485.5
25K*4MB Invalidation	1.33	21.9
25K*4MB Regeneration	2.37	46.5

1) Memory Node Component for Chunk Protection. FineMem utilizes Memory Windows (MWs) [14, 53] in RDMA for generating and regenerating rkeys. Unlike Memory Region (MR) registration, which incurs high latency during rkey creation and consumes significant on-chip resources [62], MWs operate on pre-registered MRs, focusing solely on access control through rkey generation. We found this approach to be resource-efficient. In our evaluation using a ConnectX-6 RDMA NICs, MWs support up to 16 million entries (64GB for 4KB chunks), and generating a MW rkey for a 4MB chunk takes only 1 microsecond—compared to hundreds of microseconds for MR registration, as shown in Table 1. FineMem overcomes the single-NIC 64GB limitation by generating multiple virtual functions (up to 128 VFs on a single ConnectX-6 NIC), with each VF assigned 16M memory windows.

MWs are pre-generated for each chunk/span/section once the MR is initialized, with the corresponding rkeys (main rkeys) stored in the capability table. To reduce the overhead of rkey regeneration during deallocation, the memory node pre-generates backup rkeys for each chunk/span/section. A pair of main rkey and backup rkey requires only 8 bytes, enabling the deallocation process to use an one-sided CAS operation to replace the main rkey with the backup rkey as part of the critical-path regeneration. This design ensures there is no critical path dependency on the memory node CPUs. The memory node then asynchronously scans the capability table, invalidates old rkeys, and generates new ones.

2) Compute Node Service Layer for Metadata Protection. FineMem introduces an allocation service on compute nodes to manage privileged operations related to metadata. On compute nodes, only the pre-initialized service has the authority to (de)allocate remote memory. All (de)allocation requests are handled and executed by this service.

Like many software virtualization techniques, FineMem's service process faces the challenge of API interception performance costs. To address concurrency, FineMem assigns a worker thread to each CPU core, with synchronization handled via shared semaphores. Callers write allocation requests to shared memory, wake up the allocation worker thread, and then wait for the allocation to complete. In our evaluation, this inter-process communication (IPC) introduces an overhead of approximately 2-10 µs—a reasonable trade-off compared to the significantly higher latency (10× to 100×) of performing fine-grained memory allocation directly on the memory node.

3) Protection Against Subversion. Finally, FineMem is de-

signed to resist subversion attempts across two key areas: API protection and binary protection.

API Protection. The service API exposes strictly controlled interfaces: malloc and free. To defend against malicious API calls, malloc operations are limited by per-system allocation size quotas, and free operations verify rkey correctness to prevent unauthorized memory releases.

Binary Protection. The allocation service enforces external private key authorization, managed by DM pool administrators, to prevent service forgery. Consequently, memory nodes establish memory management connections only with trusted allocation services through this authorization. In multi-tenant environments (e.g., public clouds), where malicious tenants might attempt to compromise allocation service keys, FineMem can leverage additional security mechanisms provided by the cloud provider. For example, deploying the allocation service at the hypervisor layer (e.g., FreeFlow [30]) ensures it remains trusted and isolated from tenants.

5 Porting Systems to FineMem

We have integrated FineMem with three types of DM systems: a user space memory malloc system that can allocate/free objects from remote memory (FineMem-User), a DM-based distributed key-value store (FineMem-KV), and a DM swap system (FineMem-Swap). These systems use FineMem's malloc(size) and free(addr) functions to allocate/free remote memory (flexible sizes, starting from 4KB chunk).

1) FineMem-User. FineMem-User is a memory malloc system that allows applications to allocate objects (e.g., bytes) from remote memory. FineMem-User bases on mimalloc [36] but manages space in DM (using FineMem). For object-size allocations, FineMem-User handles them entirely on the compute node. On slow-path when there are no local reserved chunks, it requests/frees chunks from FineMem and subdivides them into smaller slabs, similar to how (local) allocators operate (like tcmalloc/jemalloc/mimalloc [19, 36, 60]).

2) FineMem-KV. Key-value systems are popular systems for DM, and their remote memory management can be replaced with FineMem's allocation system. We re-implemented the slow path of FUSEE's block allocation [58] using FineMem, while keeping other components unchanged. This modification required only 300 LOC. Furthermore, unlike FUSEE, which requires memory node block servers, FineMem-KV becomes a completely one-sided kv-store system.

3) FineMem-Swap. FineMem can also be used to optimize DM memory utilization in swap-based [3] systems. FineMem-Swap includes three major components: i) an allocator in the kernel that provides APIs for allocating/deallocating remote pages (*page manager*) based on FineMem's chunks; ii) a mapping between local swap offsets and DM memory addresses (*remapper*), enabling on-demand binding of remote pages to swap entries; and iii) FineMem allocation on the page's swap-out path and deallocation after swap entries invalidation.

6 Evaluation

We characterize FineMem performance and overheads, and evaluate how well DM systems running with FineMem compare to state-of-the-art designs. Specifically, we answer the following questions:

- **Allocation efficiency:** How does FineMem reduce memory allocation latency compared to existing designs? What are the overheads associated with providing memory isolation and crash recovery?
- **Application-level impact:** How do memory-malloc-based applications perform when using FineMem-User for critical-path object allocation? How does FineMem-User adapt to applications with varying allocation size distributions?
- **System-level integration:** What is the end-to-end performance of representative DM systems (e.g., key-value stores and swapping systems) when backed by FineMem? How well does FineMem support memory pool sharing across multiple DM systems?

6.1 Experiment Setup

Testbed. We use a Cloudlab [17] cluster with 16 compute nodes and 1 memory node, which is a reasonable setting in datacenter memory pool [38]. Both compute and memory nodes use Intel Xeon 8360Y CPUs, the Mellanox ConnectX-6 100Gb NIC, and have 256GB of memory. Applications run on compute nodes and can use up to 32 cores, while we limit memory nodes to only use a single core. On each node, we run Ubuntu 22.04 with 5.15 Linux kernel and the Mellanox OFED 5.8 RDMA driver.

Implementations and Baselines. We implemented FineMem in C++ (8.5k LOC). We also implemented a mimalloc-based user-space allocation library, the FineMem-User (1.5k LOC for adding DM fine-grained allocation) that uses FineMem DM allocator interfaces, and ported FastSwap [32] to use FineMem interfaces with the FineMem-Swap (0.7k LOC for adding new remapper and chunk manager components in the kernel module).

We compared FineMem to three state-of-the-art baselines: Premmap-One-sided (from CXL-SHM [70], search based on free array), Premmap-RPC (from FUSEE [58] and Patronus [68], search based on memory node side allocator memkind [9]), and OnDemand-RPC (fine-grained allocation with registration). For fair comparisons, we ported these approaches within the same artifact as FineMem. For both Premmap-One-sided and Premmap-RPC, we statically pre-mapped 200GB of memory to avoid runtime MR registration. To minimize on-chip resource consumption, FineMem uses a single MR with pre-mapped hugepages [15, 45].

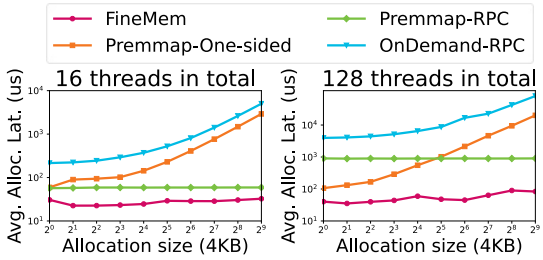


Figure 9: Allocation Performance vs. Varying Request Sizes.

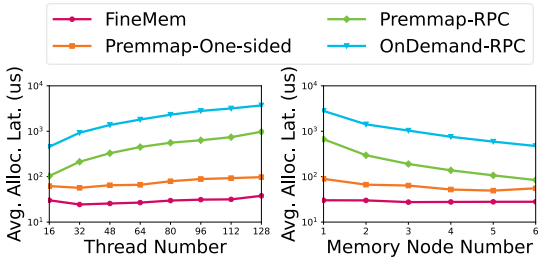


Figure 10: Allocation Performance vs. Varying Client Threads (left) and Varying Number of Memory Nodes (right). 4KB allocation granularity. Allocations distributed across multiple memory nodes using a round-robin approach.

6.2 Remote Allocation Performance

We characterize FineMem's remote allocation performance from three perspectives. First, we use allocation microbenchmarks to evaluate FineMem's performance across different allocation granularities and scalability. Second, we analyze FineMem's allocation latency, breaking it down into the various factors introduced by its design choices. Finally, we examine the memory-node-side CPU and memory overhead, as well as FineMem's crash recovery capabilities.

Microbenchmark. We utilize a Redis-based kv-store microbenchmark derived from the memory efficiency unit test in the Redis [50] test suite. This benchmark performs memory block allocations and deallocations ranging from 4KB to 2MB on remote memory. The workload initially allocates 64K blocks (totaling 256MB to 128GB) of memory, then randomly deallocates 50% and allocates another 50%, repeating this pattern. We run 500 iterations of this allocation microbench to achieve about 1 million allocations in total. Fig.9 and 10, along with Table 2, demonstrate FineMem's performance advantages over baseline approaches across various allocation granularities, levels of client parallelism, and memory node configurations. All benchmarks are executed ten times to obtain stable and converged performance results.

As shown in Fig. 10 (left), FineMem outperforms the RPC-based approaches (both OnDemand-RPC and Premmap-RPC), particularly under high client parallelism, where the RPC-based methods quickly become bottlenecked by the limited compute power of memory nodes. With a 4KB allocation granularity, we observe that the RPC-based remote allocation performance does not scale after 16 client threads. In contrast, FineMem maintains consistently low latency due to its one-sided RDMA design.

Table 2: Average/Max number of CAS retry statistics for allocation and average/P99 allocation latency. 4KB allocation granularity, 512 client threads in total.

Average / Tail	FineMem	Premmap-One-sided
Latency (µs)	43.215 / 79.347	763.01 / 16143.516
CAS Retry Times	1.333 / 142	45.108 / 20637

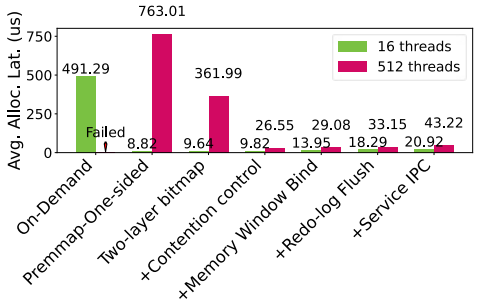


Figure 11: Factor Analysis.

FineMem outperforms existing one-sided approaches (Premmap-One-sided, without runtime MR registration) across varying allocation granularities and levels of client parallelism, as shown in Fig. 9 and 10. This performance advantage is attributed to FineMem's optimized metadata design and contention control mechanism, which significantly reduces the number of per-allocation RDMA CAS operations compared to the simple direct array design employed by existing one-sided approaches.

In the microbenchmark workloads with fragmented remote memory, the direct array metadata often requires multiple CAS operations to search for available chunks for a 4KB allocation. The performance gap becomes even more pronounced for large allocations (Fig. 9), where each chunk in a large request requires separate CAS operations.

As shown in Table 2, the Premmap-One-sided approach can require an average of 45 CAS operations per allocation, whereas FineMem requires no more than two CAS operations on average. In Fig. 11's 512-thread workload, we further drill-down the effects of FineMem's two-layer bitmap design and contention control mechanism. The two-layer bitmap reduces allocation latency by approximately 52.5% compared to the array-based approach, while the contention control mechanism contributes an additional 44% reduction by minimizing CAS retries caused by heavy contention. Combined, these two designs reduce allocation latency by approximately 95%, demonstrating the effectiveness of FineMem's design.

Factor Analysis. We evaluated the latency factors across different design components under both general cases (16-threads workload) and heavy workloads (512-threads workload), as shown in Fig. 11.

FineMem demonstrates stable latency across both general and heavy workloads. Although its chunk search mechanism involves two layers, the bitmaps are cached locally, resulting in only a slight increase in average latency compared to the

Premmap-One-sided method in 16-threads. The rkey capability introduces an additional RDMA read to retrieve the key, and the redo-log requires one RDMA CAS to flush, with each step adding approximately 5 µs. Additionally, the service layer contributes an additional 2-10 µs due to IPC overhead.

Overall, FineMem reduces allocation latency by approximately 95% compared to traditional registration-based fine-grained methods. The overhead introduced by isolation and redo-log management is minimal, accounting for only 2.5% of the registration cost.

Overhead of Pre-allocated Memory Pools. FineMem uses pre-allocated memory pools with fine-grained memory windows, which introduces initialization overhead and access latency through two primary factors. First, the pre-generation of memory windows incurs a non-trivial setup cost. Our measurements on ConnectX-5 devices indicate that creating each memory window takes approximately 166 µs. However, this overhead can be substantially reduced via parallelization: leveraging 8 threads lowers the per-window generation time to 32 µs. Second, access latency stems from on-chip cache resource contention [15, 31]. In a 32-thread experiment with randomized read to pre-generated windows (each window sequentially read in 64-byte increments), we observe that read latency increases modestly—from 4.11 µs to 4.14 µs—as the number of windows scales from 1K to 1M. Correspondingly, the impact on read/write throughput remains negligible. Importantly, recent industry efforts such as Husky [31] indicate that future hardware and software advancements have the potential to further reduce both memory window initialization overhead and contention-related access latency.

Overhead on Memory Nodes. FineMem runs two services on each memory node: the rkey regeneration thread and the recovery thread, co-running on a single core. The rkey regeneration thread scans every 100ms interval, spending 15ms for 100GB memory space (25 million 4KB chunks), and resulting in approximately 15% of one CPU usage. The regeneration speed is fast, as mentioned earlier, with 1 million memory windows regenerated per second. In our evaluation, with a backup rkey to hide the scan and regeneration time, the peak free latency remains below 1ms. The recovery detection thread wakes up every second, which has negligible overhead. Furthermore, FineMem stores external data structures at memory nodes, consuming up to 0.4% of memory (mainly caused by an 8B rkey and an 8B redo-log for each 4KB chunk), which amounts to about 400MB for every 100GB of memory.

Recovery Capability. We show FineMem's recovery ability in out-of-date detection, recovery speed, and recovery influence. 1) We ran the microbench at 1GB memory space (a narrow space with frequent malloc/free) and found that FineMem's timestamp overflow happens after about 50K memory allocation requests. As a result, updates delayed less than 1s (each allocation 20 µs) can be detected by FineMem's 7-bit timestamp, which is enough to co-work with normal RDMA timeout settings. 2) The recovery latency is highly related to memory window rebinding time, and the recovery speed is similar to memory window regeneration speed, about 1M allocation entries (chunk/span/section) per second. 3) The recovery process only has contention possibilities at span/section's bitmaps, and the influence is similar to an external thread's allocation competition, which can be handled by FineMem's concurrency design with minimal influence.

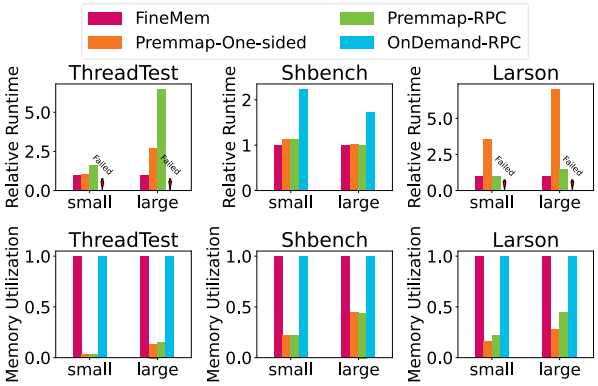


Figure 12: FineMem-User's Performance and Memory Utilization on Allocation Benchmarks.

In summary, even under heavy workloads, FineMem delivers scalable, low-latency, and predictable fine-grained memory allocation. This enables DM systems to achieve high memory utilization while maintaining strong performance.

6.3 System Case Studies

Memory Malloc System's Allocation Critical-Path. We first use the FineMem-User to demonstrate the end-to-end application performance improvement on the allocation critical path by running FineMem and other baselines with 2MB-chunk-based mimalloc on object-grained benchmarks.

1) ThreadTest [6] is a widely used allocator benchmark, creating a series of threads and repeatedly malloc/free objects. We set 256B as a small work-set (the median size of in-memory kv [69]) and 4KB (the basic OS page size) as a large work-set; 2) Shbench [44] is a stress test where some of the objects are freed in a usual LIFO order, but others are freed in reverse order. We set 8B-1KB as a small work-set, and 256B-4KB as a large work-set; 3) Larson [33] simulates a server workload using 100 separate threads which each allocates and frees objects but leaves some objects to be freed by other threads. Work-set the same as Shbench. We ran all benchmarks with 16 compute nodes, each node 32 threads.

The results, as shown in Fig. 12, highlight the performance and utilization trade-offs faced by state-of-the-art approaches:

Premmap memory waste. Premmap-One-sided and Premmap-RPC exhibit low memory utilization, even on the Shbench-large. We set 80GB memory for the premmap baselines, which is the peak memory usage of Shbench-large. Since their memory cannot be fully shared or utilized, the average memory utilization remains below 50%.

On-demand registration cost. OnDemand-RPC suffers from poor performance due to runtime memory registration. They cannot even finish some of the benchmarks (timeout).

Allocation performance issues over network. RPC baselines encounter a high concurrency bottleneck when running ThreadTest allocation benchmarks. Meanwhile, the one-sided baseline performs poorly on the Larson benchmark, where frequent thread creation and destruction in each iteration cause new threads with the Premmap-One-sided method to repeatedly start searches from the first chunk of the array.

FineMem can fully use the shared memory pool's free chunks, the same as fine-grained allocation with registration, improving memory utilization of state-of-the-art with static pre-mmap by 2.25× to 2.8×. And FineMem achieves the best performance by effectively fine-grained memory allocation in DM. It mitigates the scalability issues seen in RPC-based methods while providing fast and stable searches through its two-layer bitmap design.

Malloc System with Mixed Size Allocation Requests. To evaluate FineMem with applications that issue mixed-size allocations, we capture slow-path memory allocation system calls from various allocator and benchmark combinations.

We use the collected allocation system call sequences as requests to DM, preserving both the arrival time patterns and allocation size distributions. We selected four modern allocators: jemalloc [19], mimalloc [36], tcmalloc [60], and ptmalloc [20]. Benchmarks are chosen from the widely-used mimalloc-bench suite [13], including Shbench [44], Larson [33], Rptest [28], and Lean [1].

The size distributions of allocator and benchmark combinations, as shown in Fig. 13, vary significantly. Jemalloc and ptmalloc primarily use 4KB as the basic allocation size, tcmalloc primarily uses 2MB hugepages, and mimalloc has a median size of 64KB for small object allocation. These differences create diverse workloads with mixed allocation sizes for evaluation. To ensure trace randomness between nodes, we collect traces separately for each node and then execute them across 16 nodes, each running 8 threads.

As shown in Fig. 14, under high concurrency and mixed-size distribution, both one-sided and RPC methods experience high average latency. Besides the CPU bottleneck of RPC methods, Premmap-One-sided performs worse at jemalloc and ptmalloc. Because there are frequent 4KB chunk allocations competing with 2MB allocations, the Premmap-One-sided method finds it hard to locate an available 512 contiguous chunks in this fragmented chunk space.

In contrast, FineMem's two-layer bitmap design effectively mitigates competition between different allocation sizes and accelerates the search for power-of-2-aligned free space. It consistently maintains latency below 100 µs, with minimal impact on execution time, demonstrating FineMem's stability under high-concurrency, mixed-size allocation workloads.

Key-value System. We conducted a comparative performance analysis of FineMem-KV against different baselines

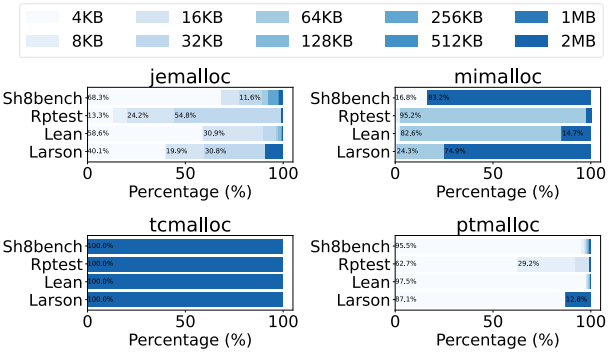


Figure 13: Mixed Size Allocation Traces' Distribution.

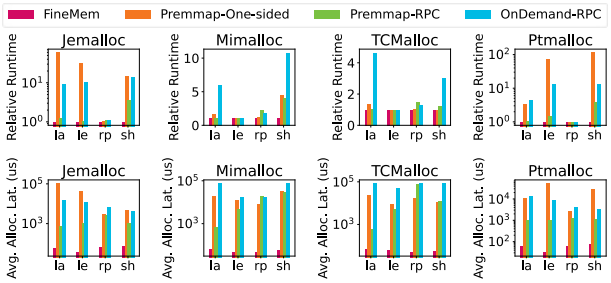


Figure 14: Performance on Mixed Size Allocation Traces.

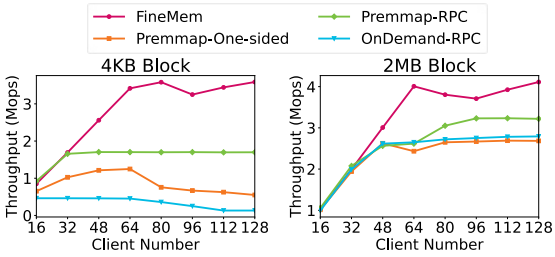


Figure 15: FineMem-KV's Throughput on YCSB-A Workload (search:update=50:50).

using standard YCSB workloads [12] (A, B, C, and D). During these tests, the client writes new key-value pairs to remote memory on each update, while allocating (or caching) multiple objects at once to reduce allocation frequency within the update transaction. For these experiments, we set the cache chunk size to 4KB (for memory efficiency) and 2MB (for better performance). Each key-value pair is 512 bytes in size.

As shown in Fig. 15, under the YCSB-A workload, which is update-intensive with approximately 50% update operations, FineMem-KV demonstrated a bandwidth improvement of around 27% to 110% over the best case of Premmap-RPC. However, in the read-intensive workloads of YCSB-B, C, and D, the performance gains were less pronounced, so we do not present those results here. FineMem's optimizations are particularly effective in enhancing FUSEE's update operations, likely due to the nature of out-of-place updates, which involve allocating new subblocks followed by freeing the old ones.

With FineMem's low latency for fine-grained allocation, using smaller block sizes like 4KB becomes more attractive. It can achieve high bandwidth similar to 2MB blocks while also

Table 3: Exclusive Fragmentation in One Compute Node.

Time(s)	1000	2000	3000	4000
Local Free Space(GB)	0	0	0	0
DM Free space(GB)	2.14	16.9	18.84	18.84
Waiting List	Snappy(33.2GB) & Redis(31GB)			

reducing internal fragmentation, resulting in a 45% reduction in memory cost compared to the 2MB block setting, as shown in the motivation section Fig. 2(a). Moreover, when testing with larger block sizes, high allocation latency still negatively impacts throughput, as shown in Fig. 15 (right), highlighting the performance advantages of FineMem.

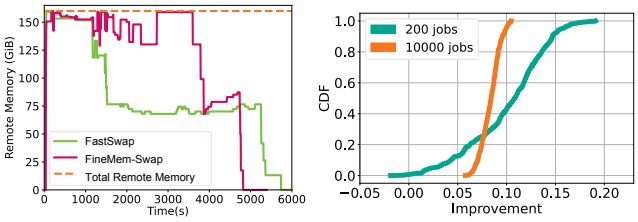
In summary, by optimizing out-of-place update operations, kv-store using FineMem resolves the dilemma between efficiency and overhead, achieving both high throughput and reduced fragmentation through fine-grained allocation.

Swap System. This evaluation focuses on comparing the memory efficiency of static pre-mapped swap systems (e.g., FastSwap) with the fine-grained dynamic swap system (FineMem-Swap). The swap systems were tested on a 7-machine testbed: 5 compute nodes with 80 GB of local memory and 2 memory nodes providing a total of 160 GB of shared DM. For FastSwap, each compute node had 32 GB exclusive memory on the DM.

Three widely deployed applications were selected for evaluation: XGBoost [11], a regularized gradient boosting framework; Snappy [21], a (de)compression library, used to compress enwiki articles [66]; and Redis [50], an in-memory database, subjected to YCSB [12] workloads. We inherited the DM-aware scheduler and simulator used by FastSwap.

We evaluated DM swap systems by placing 200 jobs with a workload ratio of XGBoost:Snappy:Redis=2:2:1 on the testbed. Compute nodes make best effort to use local memory and swap data to DM when local memory is insufficient. During system running, we collected placement data from FastSwap, and several representative states are shown in Table 3. After the 2000th second, this node had more than 50% free space in the reserved DM area, but the job queues were filled with Snappy and Redis jobs that could not be placed. As a result, FastSwap utilized only 41.39% of the total remote memory, as shown in Fig. 16 (a). In contrast, FineMem-Swap allows clients to share the DM with fine-grained allocation, resulting in an average memory utilization of 74.06%. This higher memory usage in FineMem-Swap led to a 17.71% improvement in job throughput.

To further investigate the throughput improvement in different workloads, we randomly generated 500 small workloads (each consisting of 200 jobs across 5 compute nodes sharing 160GB of DM) and 500 large workloads (each with 10,000 jobs across 40 compute nodes sharing 1280GB of DM), with varying application ratios. The CDF of FineMem-Swap's throughput improvement over FastSwap is shown in Fig. 16(b). FineMem-swap outperforms FastSwap in throughput across both small and large workloads, achieving an average throughput improvement of 8.38% to 10.69% . Therefore, FineMem improves job throughput by overcoming the static pre-mapped memory fragmentation.



(a) Remote Memory Usage. (b) Jobs Throughput Improvement

Figure 16: FineMem-Swap's Remote Memory Utilization and Job Throughput.

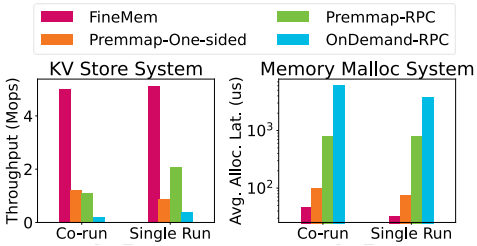


Figure 17: KV-Store System and Memory Malloc System Co-running vs. Single Running.

Overall, with support for fine-grained, on-demand allocation in the swap system, FineMem-Swap improves job throughput compared to static pre-mapped swap systems.

Memory Pool Sharing Across Systems. We conclude by evaluating the sharing of memory pools across multiple systems through a dual-workload experiment. In this setup, 8 compute nodes (16 threads each) run a kv-store system under YCSB-A workloads (in Sec. 6.3), while another 8 dedicated nodes (16 threads each) execute a memory malloc system with microbenchmarks (in Sec. 6.2) to simulate allocation-intensive scenarios and assess cross-system interference.

Fig. 17 illustrates FineMem's ability to maintain performance isolation in disaggregated memory systems. During co-execution, both kv-store throughput and allocation latency remain stable. In contrast, Premmap-RPC experiences a 46.8% reduction in bandwidth and an increase in allocation latency due to RPC core contention on memory nodes. OnDemand-RPC shows similar results with RPC contention. While Premmap-One-sided shows better tolerance for sharing, it still suffers a 75.5% reduction in bandwidth and 2.1× higher latency than FineMem, primarily due to its inherent address space search overhead. These results demonstrate that FineMem facilitates high-performance memory pool sharing in multi-system disaggregated memory environments.

7 Discussions and Limitations

While FineMem enables efficient fine-grained remote memory allocations, it does have certain limitations that we believe

are worth exploring in future work.

First, FineMem primarily focuses on scalability, isolation, and memory efficiency in DM management, along with support for compute node crash consistency. It does not address other critical issues such as cache coherence [34] or memory duplication across multiple memory nodes [35, 71]. Integrating FineMem with solutions to these aspects of DM memory management is a potential area for future research.

Secondly, FineMem pre-registers the entire remote memory, requiring DM systems to first communicate with a designated server process on the memory nodes to establish an RDMA connection. Moreover, FineMem did not optimize data-path access performance; therefore, its RDMA access performance represents a standard baseline without additional overhead or specialized optimizations.

Thirdly, FineMem focuses on RDMA-connected DM and leverages RDMA-specific mechanisms (e.g., RDMA memory windows) for isolation. FineMem's reliance on RDMA features can be technically mapped to emerging CXL-based DM. The CXL specification enables direct memory access through load/store instructions. As a fully one-sided allocator (Sec. 4.1), FineMem can seamlessly replace remote memory access with CXL memory access. Furthermore, CXL memory can be configured as globally shared memory and isolated between applications using memory protection keys (MPK), as demonstrated by Skyloft [29] and uProcess [39]. However, CXL memory presents differences in mechanisms, including memory caches [40, 70] and MPK bindings [29, 39]. Consequently, FineMem requires additional design efforts to be fully adapted to CXL, which will be addressed in future work.

Lastly, the allocation services in FineMem, which use IPC to intercept remote memory allocations, introduce overheads on the allocation path. While we believe that these overheads are justified by the security and isolation benefits provided, we are actively working on optimizing the IPC process. Future optimizations may include approaches such as kernel acceleration to further reduce these overheads.

8 Related Work

Transparent Disaggregation. To enable applications to use DM without modification, many studies leverage the swap mechanisms in kernel to utilize DM [3, 10, 22, 23, 57, 64]. For instance, Leap [2] improves the swap system's page prefetcher to better adapt to DM; Canvas [64] focuses on optimizing swap entry allocation. However, all these approaches use pre-registration strategies, leading to significant memory waste.

Native Disaggregation. Certain scenarios, such as memory malloc systems [15, 54, 59, 62, 70] and kv-store systems [41, 58, 73], allow programs to utilize DM through explicit APIs. However, many DM-native systems rely on RPC-based allocators [15, 54, 59], which do not perform well under high concurrency. Additionally, one-sided access solutions, such as CXL-SHM [70], don't provide isolation between applications.

Hybrid Usage of Transparent and Native Disaggregation. FDP-DC [55] envisions the development of a fully disaggregated and programmable data center, where applications of both transparent and native disaggregation abstractions coexist within a unified disaggregated memory pool. Furthermore, studies such as Mira [23] and Atlas [10] have explored hybrid DM data planes that combine the object-granularity of DM-native systems with the page-granularity of swap-based systems. FineMem operates under the premise that multiple applications and systems—spanning both transparent and native approaches—share a common DM memory pool.

Memory Allocator. User-space allocators [19, 25, 37, 42, 60] are widely used in software systems to enhance efficiency and performance in memory management. FineMem learns from their design philosophy and further considers the characteristics of DM architectures. LLFree [67] is a high-concurrency frame allocator in kernel, which also maintains a two-layer metadata but prioritizes exclusive entries. While FineMem adopts a sharable entry to optimize memory space efficiency.

Disaggregation Memory Isolation. For RDMA-based DM, MIND [34] uses a programmable switch, and Clio [24] uses a customized FPGA to replace the heavy MR-based isolation of RDMA NIC. To achieve both performance and easy promotion, FineMem chooses MW mechanism that comes with commercial RDMA NIC productions. Patronus [68] also uses MWs for memory protection within a single application, and it relies on RPC for memory allocation, which performed poorly in our evaluation.

9 Conclusion

We present FineMem, an efficient fine-grained remote memory management system using one-sided RDMA. FineMem removes memory region registration costs for disaggregated memory while ensuring isolation and protection through memory windows and per-compute node allocation services. FineMem features scalable and efficient metadata designs for remote chunk allocation and crash consistency mechanisms to handle compute node failures. FineMem reduces remote memory allocation latency by up to 95% compared to state-of-the-arts, enabling disaggregated memory systems to achieve low memory waste with minimal performance overhead.

Acknowledgments

We thank the shepherd and anonymous reviewers for their comments. This work was supported in part by NSFC (62472392, 62172382) and the Youth Innovation Promotion Association CAS.

References

[1] Lean. https://github.com/leanprover/lean3, 2024. [Online; accessed 1-December-2024].

[2] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *USENIX ATC 20*, pages 843–857, 2020.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Eurosys 20*, pages 1–16, 2020.

[4] Dotan Barak. *ibv_reg_mr, ibv_reg_mr_iova, ibv_dereg_mr - register or deregister a memory region (MR)*, 2006. Linux Programmer's Manual.

[5] Dotan Barak, Majd Dibbiny, and Yishai Hadas. *ibv_post_send(3) - post a list of work requests (WRs) to a send queue*. RDMA Core Userspace Libraries and Daemons, 10 2006. Accessed: 2024-01-12.

[6] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[8] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[9] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[10] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. A tale of two paths: Toward a hybrid data plane for efficient Far-Memory applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 77–95, Santa Clara, CA, July 2024. USENIX Association.

[11] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *SIGKDD 16*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[12] Brian F. Cooper. Yahoo! cloud serving benchmark (ycsb). Accessed: 2022-02-22.

[13] Daanx. mimalloc-bench: Benchmark suite to test various memory allocators. https://github.com/daanx/mimalloc-bench, 2024. Accessed: 2024-01-12.

[14] Majd Dibbiny and Yishai Hadas. *ibv_bind_mw - post a request to bind a type 1 memory window to a memory region*, 2016. Linux Programmer's Manual.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI 14*, pages 401–414, 2019.

[16] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231. IEEE, 2010.

[17] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *USENIX ATC 19*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[18] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *ASPLOS*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[20] Wolfram Gloger. ptmalloc. http://www.malloc.de/en/, 2024. [Online; accessed 1-December-2024].

[21] Google. Snappy: A fast compressor/decompressor. Accessed: 2022-02-22.

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *NSDI 17*, pages 649–667, 2017.

[23] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS 21*, pages 417–433, 2021.

[25] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *OSDI 21*, pages 257–273. USENIX Association, July 2021.

[26] Intel. 100gbe intel® ethernet network adapter e810. https://www.intel.com/content/www/us/en/products/details/ethernet/800-network-adapters/e810-network-adapters/products.html, 2024. [Online; accessed 1-December-2024].

[27] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS:Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search. In *USENIX ATC 23*, pages 585–600, 2023.

[28] Mattias Jansson. rptest. https://github.com/mjansson/rpmalloc-benchmark, 2024. [Online; accessed 1-December-2024].

[29] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. Skyloft: A general high-efficient scheduling framework in user space. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 265–279, New York, NY, USA, 2024. Association for Computing Machinery.

[30] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based virtual RDMA networking for containerized clouds. In *NSDI 19*, pages 113–126, Boston, MA, February 2019. USENIX Association.

[31] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 31–48, Boston, MA, April 2023. USENIX Association.

[32] Yash Lala. Fastswap-linux-5.16.16. https://github.com/yashlala/fastswap-linux-5.16.16, 2022.

[33] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, page 176–185, New York, NY, USA, 1998. Association for Computing Machinery.

[34] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *SOSP 21*, pages 488–504, 2021.

[35] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *FAST 22*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[36] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Report MSR-TR-2019-18 „ Microsoft, June 2019.

[37] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*, pages 244–265. Springer, 2019.

[38] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. Fast core scheduling with userspace process abstraction. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 280–295, New York, NY, USA, 2024. Association for Computing Machinery.

[40] Compute Express Link. https://www.computeexpresslink.org/. Accessed: [1-December-2024].

[41] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. SMART: A High-Performance adaptive radix tree for disaggregated memory. In *OSDI 23*. USENIX Association, 2023.

[42] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *ASPLOS 20*, ASPLOS '20, page 541–556, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *ASPLOS*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[44] MicroQuill. Smartheap technical specification. http://www.microquill.com/smartheap/sh_tspec.htm, 2024. [Online; accessed 1-December-2024].

[45] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, and Dan Tsafrir. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 97–108, 2019.

[46] Nvidia. Connectx nics. https://www.nvidia.com/en-sg/networking/ethernet-adapters/, 2024. [Online; accessed 1-December-2024].

[47] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: compacting memory management for c/c++ applications. In *PLDI 2019*, PLDI 2019, page 333–346, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *NSDI 23*, pages 181–198, Boston, MA, April 2023. USENIX Association.

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

[50] Redis. https://redis.io/. Accessed: 1-December-2024.

[51] Remote Direct Memory Access (RDMA) - Wikipedia. https://en.wikipedia.org/wiki/Remote_direct_memory_access#:~:text=In%20computing%2C%20remote%20direct%20memory,in%20massively%20parallel%20computer%20clusters. Accessed: 1-December-2024.

[52] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 01 1977.

[53] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *USENIX Security 21*, pages 4277–4292, 2021.

[54] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM:High-Performance,Application-Integrated far memory. In *OSDI 20*, pages 315–332, 2020.

[55] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 18–28, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Huijun Shen, Guo Chen, Bojie Li, Xingtong Lin, Xingyu Zhang, Xizheng Wang, Amit Geron, Shamir Rabinovitch, Haifeng Lin, Han Ruan, Lijun Li, Jingbin Zhou, and Kun Tan. Np-rdma: Using commodity rdma without pinning memory, 2023.

[57] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R Lyu. Ditto: An elastic and adaptive memory-disaggregated caching system. In *SOSP 21*, pages 675–691, 2023.

[58] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. FUSEE: A fully Memory-DisaggregatedKey-Value store. In *FAST 23*, pages 81–98, 2023.

[59] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma, 2021.

[60] TCMalloc - Google. https://github.com/google/tcmalloc. Accessed: 1-December-2024.

[61] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *USENIX ATC 20*, USENIX ATC'20, USA, 2020. USENIX Association.

[62] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *SOSP 17*, pages 306–324, 2017.

[63] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *OSDI 22*, pages 35–53, 2022.

[64] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *NSDI 23*, pages 161–179, 2023.

[65] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *OSDI 20*, pages 117–135, 2020.

[66] Wikimedia Foundation. Wikimedia downloads. Accessed: 2022-02-22.

[67] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, Daniel Lohmann, Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, and Daniel Lohmann. Llfree: Scalable and optionally-persistent page-frame allocation. In *USENIX ATC 23*, volume 65. USENIX Association, 2023.

[68] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. Patronus: High-Performance and protective remote memory. In *FAST 23*, pages 315–330, Santa Clara, CA, February 2023. USENIX Association.

[69] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *OSDI 20*, pages 191–208, 2020.

[70] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for cxl-based distributed shared memory. In *SOSP 23*, pages 658–674, 2023.

[71] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *OSDI 22*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

[72] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. Characterizing a memory allocator at warehouse scale. In *ASPLOS*, ASPLOS '24, page 192–206, New York, NY, USA, 2024. Association for Computing Machinery.

[73] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *USENIX ATC 21*, pages 15–29, 2021.

To PRI or Not To PRI, That's the question

Yun Wang[*♠] Liang Chen[*♢] Jie Ji[♢] Xianting Tian[♢] Ben Luo[♢]
Zhixiang Wei[♠] Zhibai Huang[♠] Kailiang Xu[♠] Kaihuan Peng[♢] Kaijie Guo[♢]
Ning Luo[♢] Guangjian Wang[♢] Shengdong Dai[♢] Yibin Shen[♢]
Jiesheng Wu[♢] Zhengwei Qi[♠]
[♠]*Shanghai Jiao Tong University* [♢]*Alibaba Group*

Abstract

SR-IOV and I/O device passthrough enable network and storage devices to be shared among multiple tenants with high density using virtual functions (VFs), achieving near-native performance. However, passthrough does not support page faults, requiring the hypervisor to statically pin the VM-allocated memory. This approach is unacceptable for cloud service providers (CSPs) that rely on oversubscription to enhance memory utilization and reduce costs. The Page Request Interface (PRI) was designed to support device-side I/O page faults (IOPFs) through collaboration among devices, Input-Output Memory Management Units (IOMMU), and the OS. But PRI has not seen broad adoption in devices like NICs and storage.

We propose VIO, a novel dynamic I/O device passthrough approach that achieves near-native performance and is hardware-independent. By leveraging a shadow available queue, VIO can dynamically and transparently switch devices between VIO and passthrough modes based on I/O operations per second (IOPS) pressure, balancing resource utilization and performance. Each DMA request is probed via IOPA-snooping in the virtio data plane to eliminate IOPFs, while device interrupts are directly passed through to the VM guest, enabling performance close to passthrough. VIO is extensively tested and deployed by a leading global CSP across 300K VMs, supporting both legacy and new instances while reclaiming up to the equivalent of 30K VM memory daily without compromising user Service Level Objectives (SLOs). As the scale grows, the benefits continue to increase.

1 Introduction

Cloud workloads vary significantly, ranging from simple web hosting to resource-intensive applications like large language models, all of which require high-performance I/O for optimal operation. Device passthrough is commonly used in cloud environments to provide near bare-metal performance by allowing virtual machines (VMs) direct access to hardware. However, this approach has a major limitation: Direct Memory Access (DMA), which enables devices to transfer data directly to or from memory, can target any Guest Physical Address (GPA). If the address is swapped out or otherwise inaccessible, it results in an I/O page fault (IOPF). Since most devices cannot handle such faults, DMA failures may disrupt operations or even cause system crashes. To mitigate this issue, passthrough requires static pinning of the VM's virtual memory to physical memory, which precludes memory optimization techniques [31, 44, 46] like overcommitment [15] in multi-tenant cloud environments, leading to suboptimal memory utilization and resource wastage.

In 2009, PCI-SIG addressed this challenge by expanding the Address Translation Services (ATS) specification to in-

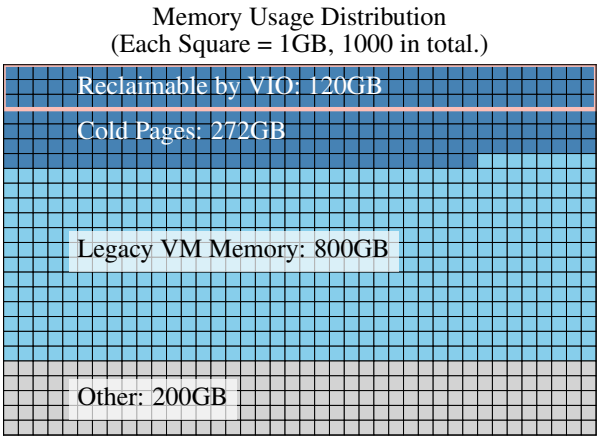


Figure 1: VMs Consumption. In a 300-node production environment, over 80% of long-running legacy (running at least one year) VMs without PRI support hold up to 800GB of memory, with $\sim$ 34% cold pages unreclaimed. With VIO enabled, approximately 120GB/day can be freed which satisfies user SLOs, translating to savings equivalent to running about 30 VMs (2C/4GB) per day.

[*]Equal contribution.

Table 1: Comparison of Memory Reclamation Methods

Method	IO-Safety	HW Compatibility	Guest modification	Reclaim type	Overhead
{v,co}IOMMU [8, 39]	✓	General	Frontend Driver	Unused + Cold	High
IOGuard [12]	✓	General	Frontend Driver	Unused + Cold	Medium
Ballooning [41]	✓	General	Frontend Driver	Unused	High
Free Page Reporting [2]	×	General	Frontend Driver	Unused	High
Hyperupcall [9]	×	General	eBPF-Tool Chain	Unused + Cold	Low
IOPF/VPRI [17]	✓	Dedicated	No	Unused + Cold	Low
VIO	✓	General	No	Unused + Cold	Low

clude the Page Request Interface (PRI) functionality [33], which enables devices to handle IOPFs. PRI relies on coordination between the Input-Output Memory Management Units (IOMMU), devices, and the operating system to manage IOPFs. Intel introduced PRI support in their IOMMUs with the 2023 Sapphire Rapids platform, significantly advancing PRI adoption despite its introduction 14 years earlier. However, delayed adoption has limited its widespread implementation, with most mainstream I/O devices—except for high-end GPUs [7, 21, 34]—currently lacking PRI support. Additionally, Linux (`v6.12`) supports PRI only in scenarios that require PASID under `IOMMU_INTEL_SVA`, further restricting its hardware compatibility [4].

Despite the potential for widespread adoption of ATS/PRI hardware and software in the future, oversubscribed cloud environments face two primary challenges. As Figure 1 depicts, in our production environment with a cluster size of 300 nodes, more than 80% of legacy VMs consume $\sim$ 800GB of memory, and this situation is expected to continue. Since the kernels of these VMs do not support PRI, hardware upgrades alone will not yield benefits. Large-scale migration to PRI-compatible devices is both technically and economically impractical. Moreover, these VMs exhibit a cold page rate exceeding 34%, suggesting an opportunity to reclaim $\sim$ 270GB of memory. Additionally, ATS/PRI introduces higher latency ($\sim 3\times$ to $\sim 80\times$) compared to CPU page fault handling due to its reliance on the PCIe bus for fault handling. This necessitates the use of an on-device buffer to store in-flight requests. The widespread adoption of SR-IOV (Single Root I/O Virtualization) technology, which virtualizes a single Physical Function (PF) of an I/O device into thousands of VFs for different guests, exacerbates scalability issues related to offline PRI and increases hardware costs [30, 40]. Therefore, PRI is not currently suitable for addressing the high-performance, high-density multi-tenant cloud scenarios.

Previous works attempt to address the challenge of statically pinning VM memory without PRI support, with solutions falling into two main categories. The first approach is software-based, including methods such as vIOMMU, coIOMMU, VProbe and IOGuard [8, 12, 39, 43]. These solutions often introduce significant performance bottlenecks, making them impractical for real-world use, and some require modifications to VM software, which is not viable in multi-tenant cloud environments. The second approach involves implementing IOPF handling at the hardware level, decoupling it from the IOMMU, as seen in solutions like On-Demand Paging and VPRI [17, 27]. However, these approaches are limited to high-end or custom hardware, restricting their broad adoption. Like PRI, these hardware solutions also introduce additional latency to I/O operations, leading to substantial performance degradation in high-performance scenarios.

In this paper, we introduce VIO (Virtual I/O), a hardware-independent, IOPF-free virtualization solution designed to deliver both high performance and low overhead in high-density, multi-tenant environments. We identify that placing page faults in the critical I/O path of complex network and storage devices is a major contributor to performance degradation. Our approach builds on the VirtIO standard, which abstracts the underlying hardware differences in VM I/O devices. VIO utilizes VirtIO's data plane to implement IOPA snooping, which detects potential page faults in advance. By eliminating IOPFs through this proactive snooping, VIO ensures high compatibility and improved performance. We implement and deploy VIO on x86 platforms, including both AMD and Intel.

As Table 1 highlights, the key differentiator of VIO is that it requires no modifications to the guest software stack—all changes reside solely in the host hypervisor. This design choice is critical for our environment, which includes hundreds of thousands of long-running VMs (running at least one year) from which memory needs to be reclaimed. The design implications and technical details are discussed further in §3.3. Prior solutions that rely on guest frontend driver or eBPF modifications are unsuitable for such deployments. Furthermore, existing approaches largely overlook the need to adapt to dynamically varying IOPS, whereas VIO addresses this gap, offering both practical engineering benefits and novel research value.

To optimize memory usage while meeting user SLOs, VIO employs dynamic device passthrough, ensuring optimal performance in high I/O operations per second (IOPS) environments. Specifically, we introduce a fine-grained, I/O pressure-based elastic passthrough strategy. Different workloads and users have varying IOPS demands, and as IOPS increase, memory requirements also tend to rise. In such cases, VIO

suspends snooping on I/O page access and directly passes the VirtIO data plane to the hardware, enabling near bare-metal performance and maximizing efficiency. When workload intensity decreases, VIO reclaims unused memory. Our experiments demonstrate that, compared to traditional passthrough, elastic device passthrough allows up to 10% daily memory reduction in production while still meeting SLO requirements.

2 Background

2.1 Trends on IO Virtualization

I/O virtualization has evolved over the years to meet the increasing demands for high performance and scalability in virtualized environments. Initially, full virtualization provided a simple and straightforward approach to I/O virtualization. In this model, devices are fully emulated through software, enabling VMs to interact with virtualized devices transparently, without requiring modifications to the guest OS. While this method offers high transparency, it incurs significant performance overhead due to frequent VM exits and data copying between the guest OS and the hypervisor.

To mitigate the performance drawbacks of full virtualization, para-virtualization [19, 24] (PV) emerged as a more efficient approach. PV reduces overhead by modifying the guest OS, allowing it to interact directly with the hypervisor through specialized drivers, rather than emulating hardware. A prominent example of PV is VirtIO, as Figure 2a illustrates, which employs a split-driver model where the guest front-end driver communicates with the host's back-end driver. This design reduces the overhead of memory-mapped I/O (MMIO) operations by avoiding unnecessary VM exits, thus improving system performance. However, as the hypervisor still manages memory and I/O operations, PV can encounter performance issues, particularly in environments with multiple concurrently running VMs.

With the introduction of technologies like VT-d, direct device assignment became feasible. This approach, particularly through SR-IOV, has gained traction due to the high-density requirements of cloud computing. As is shown in Figure 2b, SR-IOV enables physical devices to expose multiple VFs, which can be directly assigned to VMs, bypassing the hypervisor to achieve near-native I/O performance. Each VF is isolated and has its own PCI registers and interrupt handling, making SR-IOV highly effective for high-performance applications. However, SR-IOV requires vendor-specific VF drivers in the guest OS, and its lack of full compatibility across hardware vendors introduces fragmentation and device migration challenges in large-scale cloud environments.

To overcome these limitations, a hybrid approach combining SR-IOV and VirtIO has been proposed. As is shown in Figure 2c, SR-IOV provides hardware-level isolation for improved I/O performance, while VirtIO ensures compatibility and simplifies driver management. This hybrid model, known as VirtIO hardware offloading, allows VMs to access SR-IOV-enabled devices using standard VirtIO drivers, reducing vendor lock-in and facilitating seamless integration with existing software stacks. By offloading the VirtIO backend to hardware, this approach enhances both performance and scalability, offering a robust solution for modern virtualized environments.

The evolution of I/O virtualization reflects the growing demand for solutions that balance high performance with broad compatibility [29]. As virtualization technologies continue to advance, the need for efficient and scalable solutions becomes increasingly critical. VirtIO, now widely adopted as the de facto standard in virtualization environments, epitomizes this trend. As virtualization continues to evolve, the emphasis on achieving both performance and universality will intensify, with VirtIO playing a central role.

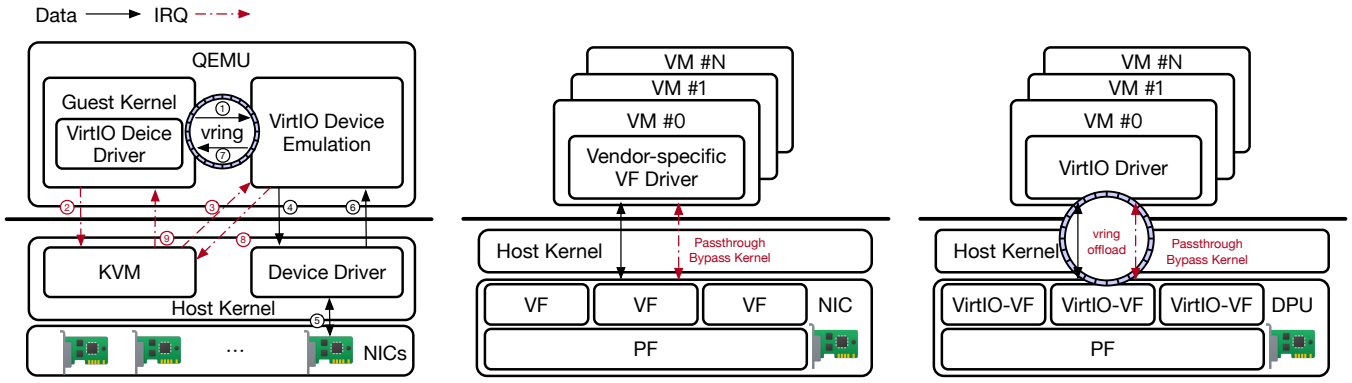
2.2 Pitfalls of IO Devices with IOPF

In traditional memory management, when the CPU encounters a page fault, i.e., when it attempts to access a virtual page not currently mapped to physical memory, an exception is triggered. The operating system's memory management unit (MMU) handles the fault by performing the necessary address translation, which often involves loading the page from disk into physical memory. Once the page is mapped, the CPU can resume its operation seamlessly.

Similarly, when an I/O device performs DMA to access a virtual memory page that has not been mapped yet, it triggers an IOPF. However, unlike the CPU, devices generally cannot handle such faults on their own. The device lacks the ability to resolve the fault [11, 28] or perform the necessary address translation. If the system fails to resolve the IOPF, it can lead to severe system instability and may even result in a system panic.

To address this issue, PCI-SIG introduced the PRI in the PCIe 4.0 specification. PRI provides a standardized interface that allows devices to request the physical address translation of virtual memory pages from the host system. With PRI, when a device encounters an IOPF, it can directly request memory translation from the operating system or hypervisor. This mechanism enables the system to resolve IOPF without CPU intervention, allowing the device to continue its memory access without causing a system crash.

Figure 3 illustrates how PRI handles IOPFs. Initially, the device's PRI function sends a page request message to the IOMMU over PCIe (①). This message is received by the PRI Server (PRS) within the IOMMU, which writes it to the PRS event queue, triggering a host interrupt (②). This interrupt asynchronously notifies the IOMMU driver, leading to significant delays. In steps ③ and ④, the memory subsystem handles the process similar to a CPU page fault: the system requests a physical page, populates it, and updates the page



(a) VirtIO (b) SR-IOV (c) VirtIO Offload

Figure 2: The evolution of I/O virtualization. (a) VirtIO illustrates the para-virtualization approach with a split-driver model to improve performance by reducing VM exits. (b) SR-IOV demonstrates hardware-level direct device assignment for near-native performance by exposing VFs to VMs. (c) VirtIO hardware offload combines the benefits of SR-IOV and VirtIO, leveraging hardware offloading to achieve both high performance and broad compatibility.

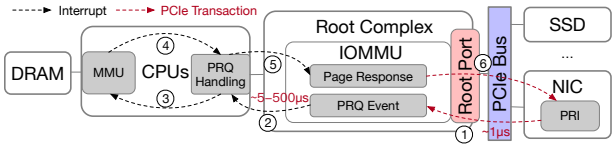


Figure 3: IOPF Handling Process. PRQ handling in step ② introduces the most significant delay, followed by memory population and response delivery.

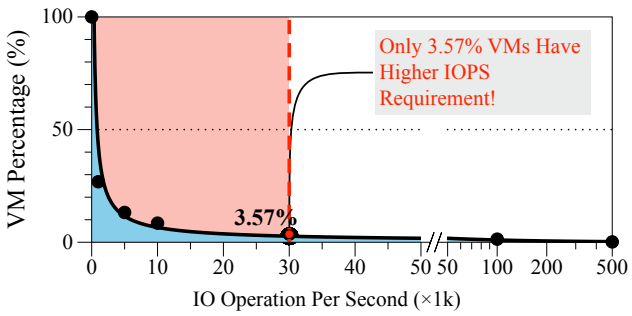


Figure 4: IOPS distribution in a CSP production environment, showing a long-tail pattern where 73.14% of workloads have IOPS below 1,000, while less than 3.57% exceed 30,000 IOPS.

tables. Finally, the IOMMU driver sends a response to the device PRI interface (⑤, ⑥).

Mellanox [20] and VPRI focus on optimizing the latency of the IOPF process, claiming that the entire IOPF handling cycle introduces a latency of a few hundred milliseconds. However, based on previous studies and our experiments, we find that because IOPF occurs in the critical path of DMA, any delays in this path can cause the device to drop the current packet. The device then relies on higher-layer protocols (such as TCP/RDMA) to handle packet loss through retransmission. This retransmission process can lead to packet damming or flooding, causing delays that are several hundred milliseconds long, $\sim$ 100 times the latency of the IOPF itself.

Moreover, another significant pitfall in the deployment of ATS/PRI arises from the need for synchronization between on-device memory and the CPU's page table information. Modern systems, especially those with large memory configurations, such as 2TB memory machines, face substantial challenges when attempting to manage this synchronization. For example, using a 4KB page size, synchronizing such a large memory footprint can lead to an enormous overhead in terms of memory storage. Specifically, the system must handle the synchronization of up to 500MB memory, which results in a considerable memory requirement just to maintain consistency across device and CPU memory page tables. This scale of memory management creates a significant burden in terms of both energy consumption and computational overhead, which, when combined with the performance penalties from DevTLB misses and the need for extensive TLB capacities, exacerbates the challenges associated with scaling ATS/PRI in production environments.

In summary, PRI-based solutions place IOPF handling in the critical I/O path, which interrupts the original flow of I/O operations, thus leading to significant performance degradation. Therefore, due to its inherent characteristics, PRI is not suitable for use in I/O-intensive environments.

2.3 IOPS Distribution in Cloud Environment

As is illustrated in Figure 4, the IOPS distribution in a CSP production environment demonstrates a distinct long-tail pattern. The vast majority of virtual machines, accounting for

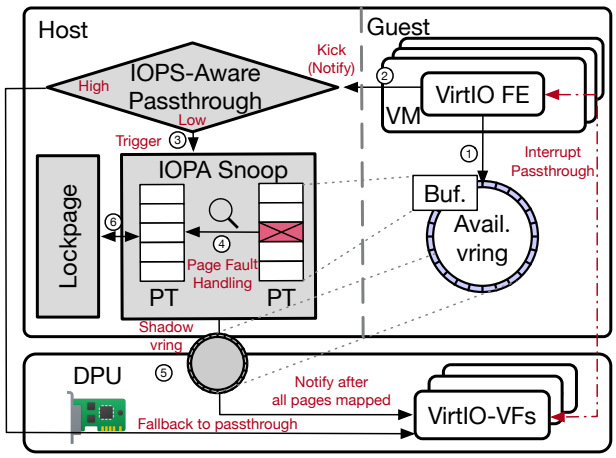


Figure 5: Overview. The figure illustrates the VIO process, where IOPF handling is offloaded to the hypervisor, transparent to both the software and hardware. This approach eliminates IOPF overhead from the critical I/O path, improving performance while maintaining full compatibility.

73.14%, have IOPS demands that do not exceed 1,000. This indicates that most workloads in the cloud operate at relatively low IOPS levels. Meanwhile, only a small fraction of virtual machines—less than 3.57%—require more than 30,000 IOPS, highlighting the rarity yet importance of high-IOPS workloads in production environments.

This distribution reflects the resource heterogeneity in cloud environments, where low-IOPS workloads dominate while high-IOPS workloads represent a smaller but critical segment. The latter are often accompanied by memory-intensive operations. In such scenarios, memory oversubscription, a common practice to optimize resource utilization, can lead to significant interference with high-IOPS workloads, disrupting performance and potentially violating SLOs.

To address these challenges, a potential solution involves an elastic passthrough strategy capable of dynamically switching between resource sharing and passthrough modes based on real-time workload demands. Under typical conditions, low-IOPS workloads efficiently share resources to maximize utilization. When workloads intensify and IOPS demands increase, the strategy shifts to direct passthrough, minimizing overhead and delivering near-native performance. By dynamically adjusting working mode based on IOPS intensity, such an approach has the potential to balance high efficiency and reliable performance across diverse and complex workloads in cloud environments.

3 Design & Implementation

3.1 Overview

Based on our analysis and observations, we propose a novel IOPF system called VIO, as is illustrated in Figure 5. VIO is a software-based, elastic device passthrough solution that adheres to the VirtIO 1.0 standard. The design of VIO enables IOPF support without relying on any specific CPU or hardware vendors, achieving this entirely through software. In contrast to the less widely adopted ATS/PRI solution, VIO is hardware-independent and transparent to the guest operating system. The key features of VIO are as follows:

1. IOPA-Snoop: VIO snoops every VirtIO command to ensure that DMA-buffered pages are handled correctly. This mechanism guarantees that page faults will not occur during device DMA operations, eliminating the need for PCIe PRI-based page requests.

2. IOPS-Aware Elastic Passthrough: As device I/O pressure increases, VIO switches to a passthrough mode. It uses a shadow available ring to enable elastic switching between device passthrough and VIO passthrough. By toggling the IOPT in the IOMMU to the shadow available ring, VIO achieves transparent passthrough mode switching for the VM. This mechanism also supports seamless upgrades to VIO for existing VMs, enabling hot upgrades without disruption.

3. Adaptive Lockpage: VIO implements device-agnostic I/O page access logging through IOPA-snoop, which collects information about IO page access pattern. Based on this information, VIO introduces an adaptive lockpage mechanism that optimizes the usage of I/O hot pages, improving memory efficiency and reducing IOPFs.

Figure 5 illustrates the VIO process as follows: **Step ①:** The VirtIO frontend in the VM prepares the buffer and places it in the available ring. **Step ②:** The VirtIO frontend then uses a kick to notify the backend that the buffer is ready for device use. **Step ③:** Depending on the current IOPS pressure, the process is divided into two paths: (a) When the VM is under heavy load, the process directly follows the passthrough fast path. (b) When the device is in VIO mode, the kick triggers an IOPA-Snoop. It is important to note that in both modes, hardware interrupts are directly passed through to the virtual machine. **Step ④:** The IOPA-Snoop thread handles any page faults in the buffer prepared in step ① by querying the Extended Page Table (EPT). **Step ⑤:** The IOPA-Snoop thread then kicks the actual hardware, initiating the I/O operation.

During hardware I/O operation, IOPA-Snoop ensures that the device can continue the operation without being interrupted by an IOPF. Compared to the PRI-based solution, this

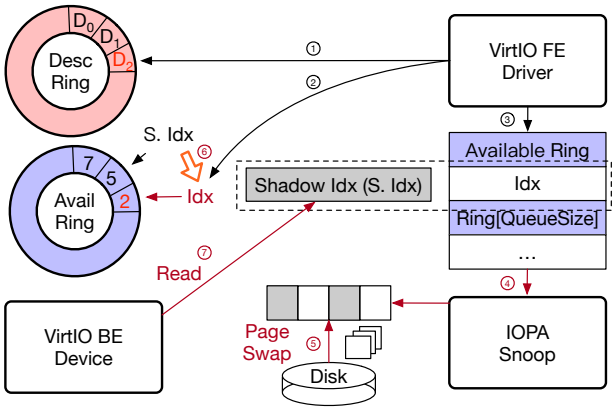


Figure 6: Overview of IOPA Snoop. This module offloads IOPF handling to the hypervisor, without device involvement. It ensures that DMA operations can continue without IOPF by guaranteeing that all pages are mapped before the device accesses them.

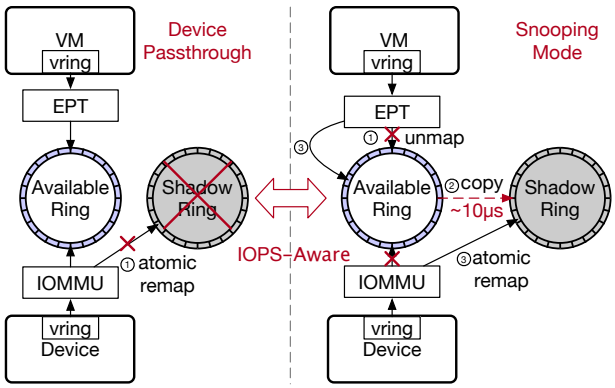


Figure 7: Elastic passthrough. It illustrates the IOPS-aware mechanism, switching between snooping and passthrough modes using shadow page tables to manage memory mappings.

approach removes the IOPF overhead from the critical path of the I/O operation.

The changes in the VIO process described above occur within the host's hypervisor, making them transparent to both the upper-level guest and the underlying hardware. Unlike PRI, the current page fault handling in VIO is no longer initiated by asynchronous interrupts. This design ensures that performance improvements in IOPF handling are achieved without requiring modifications to guest operations or additional hardware functionality. By offloading IOPF handling to the hypervisor and optimizing the I/O path, VIO enhances system performance while maintaining full compatibility between the guest and the hardware.

3.2 IOPA-Snoop Mechanism

VirtIO is a general I/O virtualization framework that allows the hypervisor to emulate a variety of virtualized devices, making them accessible within a virtual machine through API calls. It is widely used in cloud environments to enhance virtualization I/O performance and ensure compatibility across different systems. Virtqueue serves as the transport layer abstraction between the VirtIO front-end and back-end. To prevent devices from encountering page faults during DMA, VIO introduces the IOPA-Snoop module to manage the virtqueue.

Figure 10 illustrates the process by which the VirtIO front-end adds requests and notifies the back-end for consumption. **Step ①:** The front-end driver allocates a buffer and adds it to the descriptor table. **Step ②:** The front-end driver then adds a new entry with the head index of the descriptor chain, which describes the request in the available ring entries. **Step ③:** The front-end driver increments the index by the number of new entries.

In the standard VirtIO path, the back-end device eventually observes the available index and uses the buffer allocated in step ① by referring to the descriptor table based on that index. If the page in the buffer has been unmapped by the hypervisor, it triggers an IOPF, which would normally require the ATS/PRI hardware process to resolve. If the back-end device waits for the page fault to be handled before continuing, it would need to store additional information in a special buffer to manage multiple in-flight page faults. This is impractical for I/O devices due to resource limitations. As a result, most hardware implementations drop the related packets, relying on higher-layer protocols like TCP/RDMA to handle retransmission, which leads to performance degradation.

In our approach, we introduce a shadow index mechanism, where the device and driver operate on different indices. After step ③, the device sees a shadow index, which has not yet been updated and still points to the old address in the available ring. The following steps outline the process: **Step ④:** The IOPA-Snoop module detects the index update and snoops the buffer allocated in step ①. **Step ⑤:** The IOPA-Snoop module reads the unmapped data from the swap to ensure that the pages in the buffer are correctly mapped. **Step ⑥:** The shadow available index is then updated to the available index. **Step ⑦:** The back-end device observes the updated shadow available index and proceeds with the DMA operation.

In this process, IOPF handling is entirely managed on the hypervisor side, with no direct involvement from the device. Even when multiple page faults occur simultaneously, the handling process remains consistent with traditional CPU page fault handling, effectively removing IOPF handling from the critical path of the DMA operation.

3.3 Elastic Passthrough

In cloud environments, many VMs run for extended periods and may not be able to easily take advantage of new hardware features such as PRI. To ensure that both existing hardware

and software can benefit from improvements, we introduce elastic passthrough. Elastic passthrough enables VMs to seamlessly switch between device passthrough and snooping mode without requiring any changes to the VM or the underlying hardware.

To protect sensitive data from malicious VMs, the hypervisor utilizes EPT to control each VM's memory access. The VM's MMU translates Guest Virtual Addresses (GVA) into GPA, while the hypervisor maps these GPAs to Host Physical Addresses (HPA). Additionally, the IOMMU translates I/O Virtual Addresses (IOVA) to HPA, ensuring that malicious devices cannot gain unauthorized access to memory via DMA.

VIO utilizes separate page tables for the device and driver when accessing the available ring. VIO employs separate available rings: a native available ring for the device and a shadow available ring managed by the hypervisor. This design allows the hypervisor to use the shadow available ring for snooping mode, without necessitating any modifications within the VM's guest OS or applications.

Transitioning from Passthrough to Snooping: As is shown in the right side of Figure 7, when switching from passthrough mode to snooping mode, the hypervisor follows these steps: **Step ①:** Unmaps the available ring in the EPT, blocking new requests if any are in-flight. **Step ②:** Copies the contents of the available ring to the shadow available ring, which takes about 10 microseconds. **Step ③:** Atomically remaps the available ring to the shadow available ring in the IOMMU's IOPT, and then remaps the EPT back to the available ring. This sequence completes the transition to VIO mode without any noticeable changes to the VM or device.

When operating in snooping mode, each I/O request from the VM triggers an IOPA-Snoop operation. Each snooping operation introduces a small overhead (approximately 4µs according to our measurements, as detailed in the §4.1). At low IOPS, the cumulative overhead of snooping is negligible, and the benefits of preventing IOPFs outweigh this cost. However, as the IOPS increases, the cumulative overhead of these snooping operations grows proportionally, potentially becoming a significant performance bottleneck. Furthermore, we observe that VMs with high IOPS workloads tend to require most, if not all, of their allocated memory to maintain performance. In such scenarios, the benefits of memory reclamation through snooping diminish, making it more efficient to switch these heavy-load VMs back to passthrough mode.

Transitioning from Snooping to Passthrough: To mitigate the snooping overhead under heavy I/O load and to avoid unnecessary snooping for VMs that benefit little from memory reclamation, VIO incorporates an IOPS-Aware Elastic Passthrough feature. An IOPS monitor tracks the I/O activity of the VM. When the IOPS exceeds a predefined threshold (e.g., 100k IOPS in our production environment), the system automatically triggers a switch back to passthrough mode. The left side of Figure 7 illustrates this transition process. During this transition, the hypervisor remaps the shadow available ring back to the original native available ring, allowing the VM to operate in passthrough mode and bypass the snooping overhead, thereby maintaining high I/O performance. This threshold-based switching mechanism dynamically adapts to workload demands, ensuring that the system operates efficiently across varying I/O loads.

The transition from snooping mode back to passthrough mode is optimized to minimize performance disruption. Before initiating the switch, the hypervisor proactively swaps in swapped-out pages. Because snooping continuously swaps in pages, relatively few pages need to be swapped in when transitioning back to passthrough mode. This swap-in operation occurs in parallel with ongoing snooping, effectively hiding the latency of page retrieval. When the actual mode switch takes place (remapping the shadow available ring to the native available ring), all necessary pages are already resident in memory. This design choice eliminates IOPFs from the critical path of the transition, resulting in a smooth switch with no noticeable performance degradation.

To ensure that VIO can benefit existing long-running legacy VMs, we utilize a combination of Elastic Passthrough and VMM live upgrade. VMM live upgrade is a technique that has been widely adopted in both industry and academia to update the underlying hypervisor (KVM/QEMU) without requiring any reboot or reconfiguration of the VMs themselves [37]. VIO's design is implemented entirely within the hypervisor, meaning that no modifications or upgrades are necessary within the guest VMs to take advantage of its benefits.

We choose Orthus [50] to live upgrade VMM as it is widely deployed in our production. This VMM live upgrade process leverages a dual KVM approach, where a new, upgraded version of KVM can be loaded while the original KVM instance continues to run the VMs. Then, the VMs are "grafted" from the old KVM to the new KVM, and device ownership is handed over to the new VMM. This process enables a seamless upgrade of the QEMU process using a fork-exec model, where the original QEMU process forks, and the child process loads the new QEMU image. The upgraded QEMU includes VIO's Elastic Passthrough capabilities. Once the legacy VM is running under the upgraded hypervisor, its transition between passthrough mode and VIO's snooping mode operates identically to a VM that was initially started in a VIO-enabled environment.

Figure 8 illustrates the Iperf performance of a legacy VM during this upgrade process in our production environment. As the graph shows, there is some performance fluctuation during the 200ms window while QEMU is upgraded and the VM is switched to VIO mode. It's important to note that this fluctuation occurs under peak Iperf bandwidth conditions.

During a production rollout, we schedule these upgrades during periods of low workload for the legacy VMs to minimize any potential impact. Crucially, the VMs do not need to

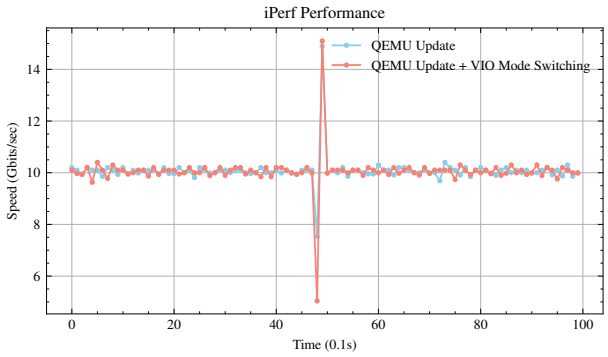


Figure 8: iPerf performance during VMM upgrade

be restarted or reconfigured, and the entire process is transparent to the guest operating system and applications. This is a significant advantage, as it avoids the complexity and potential disruption of coordinating upgrades across a large number of diverse VMs.

Overall, Elastic Passthrough allows VMs in a virtualized environment to seamlessly switch between full device passthrough and VIO modes. This flexibility ensures that memory usage is optimized and performance goals are achieved without requiring any changes to the VM or hardware.

3.4 Lockpage

The IOPA-snoop mechanism (see §4.1) ensures the safety of I/O pages involved in DMA operations. However, the repeated reclamation and snooping of specific pages can introduce additional performance overhead. To mitigate the frequency of page faults caused by snooping, VIO employs a lockpage mechanism to pin particular I/O hotspot pages, thereby significantly reducing the occurrence of IOPF page faults.

I/O operations exhibit strong temporal locality, as the same memory pages may be accessed repeatedly within short intervals. In contrast to the more random and less predictable access patterns typical of CPU operations, I/O operations generally follow consistent and repetitive access patterns. This predictability facilitates more effective optimization strategies, such as pinning frequently accessed pages to prevent their eviction.

VIO employs two primary lockpage strategies to leverage I/O page locality. A lockpage bitmap manages these locked pages, with memory managed in 2MB chunks. Additionally, VIO maintains an internal table, enabling it to bypass the checking of Page Table Entries (PTEs) for pages listed in the lockpage bitmap, prior to the IOPA-snoop process.

Static Lockpage: This strategy locks frequently used I/O pages based on their spatial continuity. For example, the VirtIO RX queue, which is preallocated with contiguous memory regions, is entirely locked to prevent its pages from being swapped out. This approach ensures that VIO does not need to verify the presence of these pages in memory before DMA operations, thereby enhancing performance. Currently, VIO employs static lockpage exclusively for the VirtIO RX queue due to its pre-allocated and contiguous memory layout.

Algorithm 1 Adaptive Lockpage Mechanism in VIO

```
1:  Initialize Lockpage Bitmap with 2MB granularity
2:  Initialize Active List and Inactive List for each Protection Domain
3:  procedure IOPA_SNOOP(access_log)
4:      for all page ∈ access_log do
5:          MOVETOACTIVELIST(page)
6:          if ISINACTIVELIST(page) then
7:              UNPINPAGE(page)
8:          end if
9:      end for
10: end procedure
11: procedure MOVETOACTIVELIST(page)
12:     Remove page from Inactive List if present
13:     Add page to the head of Active List
14:     if ISACTIVELISTFULL then
15:         EVICTFROMACTIVELIST
16:     end if
17: end procedure
18: procedure EVICTFROMACTIVELIST
19:     evicted_page ← Remove page from tail of Active List
20:     ADDTOINACTIVELIST(evicted_page)
21:     PINPAGE(evicted_page)
22: end procedure
```

Adaptive Lockpage: The adaptive lockpage mechanism in VIO is inspired by the Dual LRU and Multi-generational LRU algorithms utilized in the Linux kernel. The primary objective is to avoid pinning pages in the youngest generation (active list) since these pages are already identified as hot by the EPT through the IOPA-snoop mechanism. Pinning these pages would unnecessarily lock those that do not require it. Instead, the focus is on pinning pages from the second young/inactive list, which represent long-term recently used pages. By pinning these pages, the kernel's page reclaim module is prevented from evicting them, thereby reducing repeated page faults and enhancing overall system performance.

The adaptive lockpage algorithm (Algorithm 1) manages memory pages using `Active` and `Inactive lists`. When IOPA-snoop detects a page access, it transfers the page to the `Active List` and unpins it if it was previously in the `Inactive List`. If the `Active List` becomes full, the least recently used page is evicted to the Inactive List and subsequently pinned to retain it in memory. This selective pinning strategy ensures that frequently accessed I/O pages remain available, reducing the number of IOPFs and enhancing the ultimate system performance.

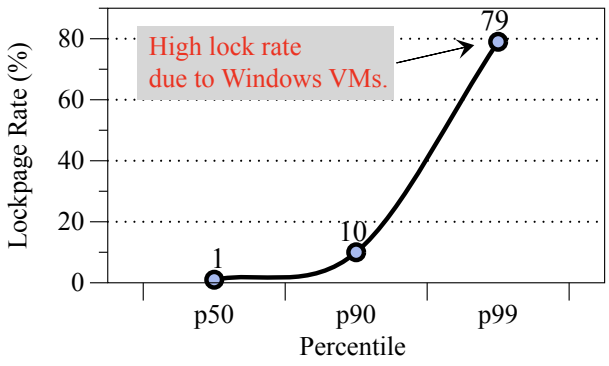


Figure 9: Lockpage rate in a production environment, showing P50 at 1%, P90 at 10%, and P99 spiking to 79% due to poor I/O locality in Windows VirtIO driver.

Lockpage Rate: We collected lockpage data from about 300K virtual machines in our production environment. As shown in Figure 9, the analysis of lockpage rates reveals significant differences in memory usage across VMs in production. The median lockpage rate is 1%, indicating strong I/O locality for most VMs. At the 90th percentile, the lockpage rate increases to 10%, reflecting the memory demands of more intensive workloads. However, at the 99th percentile, the lockpage rate jumps sharply to 79%, highlighting a subset of VMs with significantly higher memory locking requirements.

This sharp increase is largely due to approximately one-quarter of the VMs running the Windows VirtIO driver. These VMs exhibit poor I/O locality because the driver disrupts typical memory access patterns, causing a much larger portion of memory pages to be locked. This observation suggests that optimizing the Windows VirtIO driver could improve I/O locality and significantly reduce memory locking overhead. Although adaptive lockpage strategies can effectively reduce the lockpage rate, we adopt a static lockpage configuration in production for ease of maintenance.

4 Evaluation

We evaluate VIO across several dimensions through experiments. First, we examine its performance using micro-benchmarks, focusing on the end-to-end latency for handling IOPF. Next, we illustrate its performance with real-world benchmarks, comparing it to relevant hardware-based solutions. Finally, we analyze data from a production environment to verify that VIO meets the SLO requirements for users in public cloud settings.

Experimental Setup: We deployed VIO in one of the largest global CSP production environments. Due to the lack of commercially available network cards that support PRI, we chose VPRI from SOSP'24 as the baseline [17]. VPRI is a state-of-the-art hardware implementation of PRI that integrates on-device PRI functionality, enabling IOPF processing on the DPU. We have implemented VPRI ourselves, and since VIO does not rely on specific hardware conditions, we are able to support both VPRI and VIO on the same platform. Configurations are shown in Table 2.

Table 2: Environment Configuration

Type	Hardware Configuration
Host	Intel Xeon CPU Platinum 8269CY 52C/104T@2.50GHz in 2 sockets 12 * 16GB DRAM, 1TB SSD
DPU	Connection: PCIe GEN3, 8 lanes Device emulation: up to 2300 VFs Max bandwidth: 200 Gb/s
VM	4 vCPUs, 8GB RAM CentOS Linux release 7.9.2009 NIC device: dual queue virtio-net x1 10Gb/s max throughput, 190,000 PPS

Benchmarks: To demonstrate the impact of VIO, we select various I/O-intensive applications, which are highly representative of common cloud computing scenarios. The selected applications and their setups are listed as follows.

- **iperf3:** Tests network throughput using iperf3 with a single thread and a packet size of 1024 bytes. Simulates TCP-based data transfer to evaluate system bandwidth over a specified duration.[1]
- **nginx:** Measures web server performance under stress using two threads and 200 concurrent connections. Simulates heavy HTTP traffic to assess nginx's handling of simultaneous requests.[2] For Nginx benchmarks, we used 2KB static web pages.
- **Redis:** Benchmarked with redis-benchmark, using 16 threads performing GET operations on 256-byte data. Evaluates Redis's ability to handle concurrent database queries efficiently.[3] For Redis benchmarks, we used 16 connections (`-c 16`).
- **Memcached:** Tested with memaslap, using 8 threads operating on 64-byte keys (`-c 8`). Assesses Memcached's performance under concurrent GET and SET workloads.[4]

4.1 IOPA-Snooping Overhead

To evaluate the efficiency of VIO in production environments, we monitored its deployment on a sample of 10,000 VMs.

[1] iperf3, https://iperf.fr/
[2] nginx, https://nginx.org/
[3] Redis, https://redis.io/
[4] Memcached, https://memcached.org/

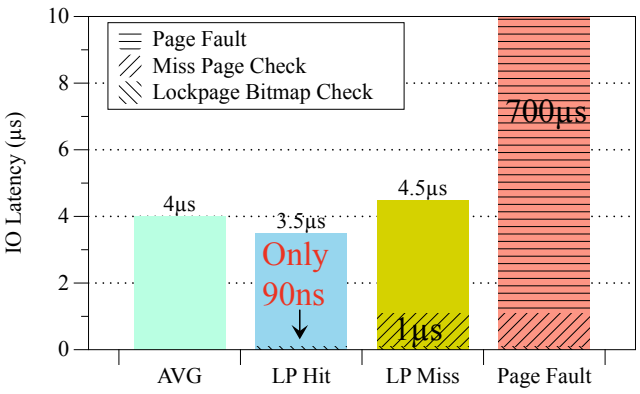


Figure 10: IOPA-Snooping Latency. The figure shows the latency distribution of the IOPA-snoop process, detailing the impact of lockpage hits (3.5µs) and misses (4.5µs) on overhead, with page faults averaging 700µs.

We sampled the IOPA-snoop process, which handles I/O page accesses in VIO. The results provide valuable insights into the latency distribution and the effectiveness of the lockpage mechanism in reducing overhead.

As is shown in Figure 10, the IOPA-snoop process incurs a average latency of 4µs, with a clear distinction based on lockpage hits and misses. When a lockpage hit occurs, the latency is reduced to approximately 3.5µs, as the system only needs to query VIO internal bitmap, which takes 90ns. For lockpage misses, the latency increases to 4.5µs, as the system must first check whether the page is present in memory, taking around 1µs. If the page is not present, a page fault is triggered, which has an average handling time of 700µs.

These findings highlight the efficiency of VIO's lockpage mechanism, which minimizes the need for costly page fault handling by ensuring that frequently accessed I/O pages are pinned in memory. Even in cases where a page fault occurs, VIO ensures robust performance with a well-contained average fault-handling latency. This efficient handling of I/O page accesses demonstrates VIO's capability to maintain low overhead and stable performance, making it suitable for large-scale cloud environments with diverse workloads.

4.2 Applications

The performance comparison between VPRI and VIO across four I/O-intensive applications, reveals differences in their ability to handle IOPF. The performance comparison between VPRI and VIO was evaluated under varying I/O pressure by injecting IOPFs at different rates (1/2/5/10 per second) and introducing diverse fault latencies. All tests were conducted under memory oversubscription conditions with Snooping mode enabled. Please note that under such high IOPS conditions, VIO would typically run in passthrough mode. However, in this case, to demonstrate the worst-case performance of VIO, we used snooping mode for comparison. These latencies are chosen to reflect the wide range of delays observed in cloud SSDs: while high-performance cloud SSDs offer read/write latencies around 4ms, others can be significantly higher. By adjusting both the IOPF frequency and latency, we simulate conditions that capture the complexity and variability of real-world cloud storage environments.

As is shown in the Figure 11, VPRI experiences significant throughput degradation as delay increases due to its reliance on retransmissions to recover from packet loss caused by IOPFs. For instance, in Redis, VPRI throughput drops by 60% under 10ms delay, while Nginx and Memcached exhibit similar degradation trends of 45% and 57%, respectively. This cascading effect of retransmissions highlights the sensitivity of VPRI to IOPF frequency, particularly in workloads with high transaction rates or network demands.

In contrast, VIO demonstrates excellent stability under the same conditions. By isolating page fault handling from the critical I/O data path, VIO avoids packet loss and retransmissions, resulting in minimal performance degradation. Across all applications, VIO's throughput reductions remain below 10%, even at higher delays. For example, in Redis, VIO achieves a modest 6% drop under a 10ms delay, while Nginx and Memcached see reductions of 9% and 6%, respectively. This consistent performance advantage underscores VIO's ability to maintain stable throughput across a range of workloads and delay conditions. Redis shows poorer performance in low IOPF situations because the frequency of snooping operations becomes a significant bottleneck due to snooping latency. When IOPS is high, total snooping overhead increases proportionally with the growth in I/O operations. This data supports the design goal of Elastic Passthrough of employing snooping in most low-IOPS scenarios and switching to passthrough in high-IOPS scenarios.

Overall, the results emphasize the effectiveness of VIO in mitigating the impact of IOPFs. While VPRI struggles with retransmission overheads that degrade performance as delays increase, VIO ensures predictable and reliable throughput by handling page faults in a non-intrusive manner. This makes VIO a robust solution for diverse cloud applications requiring stable and efficient I/O operations under varying workloads and network conditions.

4.3 I/O SLOs

In production environments, intermittent I/O jitter caused by infrastructure issues such as network congestion often results in long-tail performance degradation and packet loss. This significantly impacts I/O SLO metrics, prompting CSPs to invest substantial resources in maintaining I/O jitter within tolerable levels. However, the limitations of the current IOPF architecture exacerbate I/O jitter. Frequent IOPFs, particularly during rapid successive bursts, introduce retransmissions and delays, directly leading to I/O SLO violations.

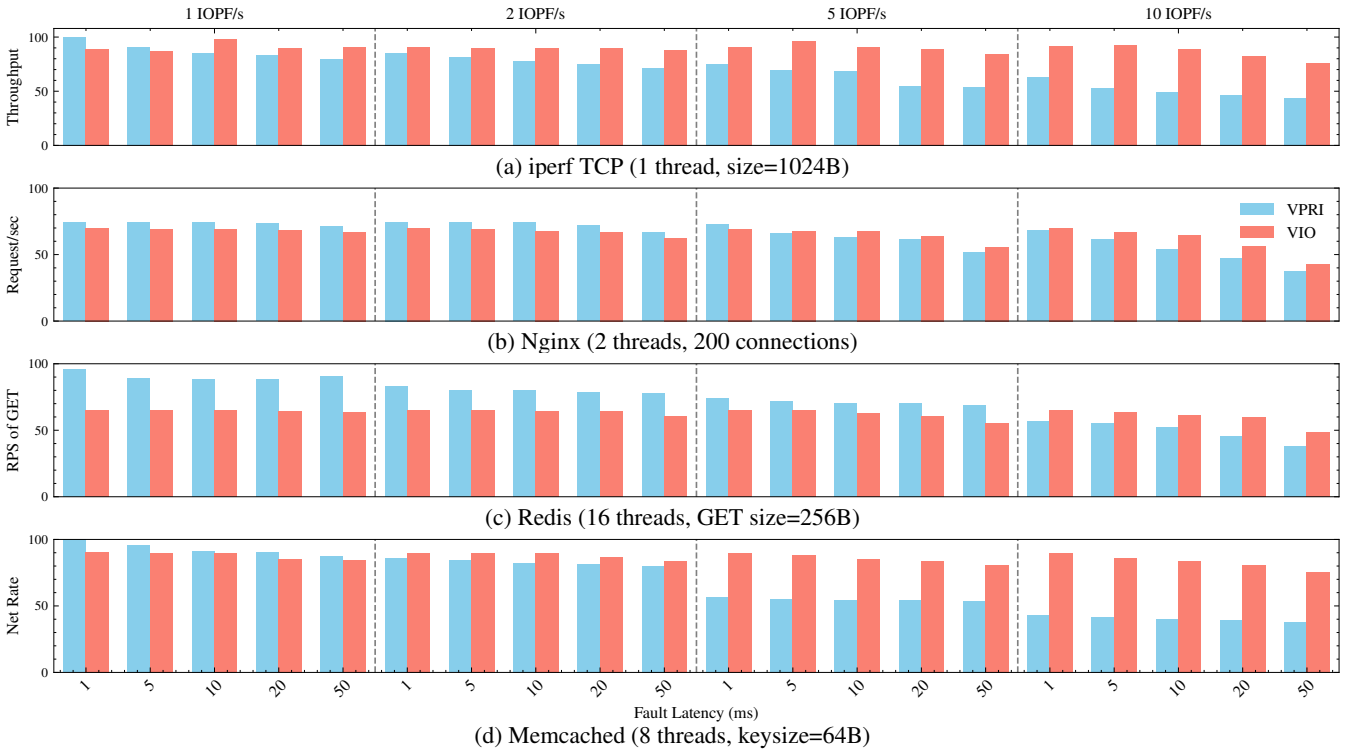


Figure 11: VPRI[SOSP'24] vs VIO Performance comparison normalized to respective passthrough mode.

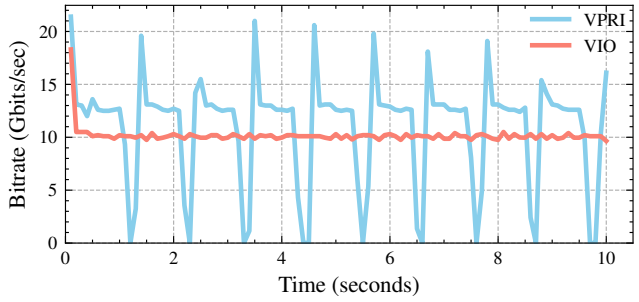


Figure 12: iperf3 performance when IOPF occurs in a fixed rate of 1s and fault latency of 5ms.

To evaluate the impact of IOPF on I/O jitter, we used iperf to measure network bandwidth, sampling throughput data every 0.1 seconds. Figure 12 reveals that VPRI suffers from significant bursts of packet loss due to its reliance on retransmissions when an IOPF occurs. This behavior leads to highly unstable throughput, frequently dropping nearly to zero during retransmission periods. These bursts not only degrade the average throughput but also amplify I/O jitter, making it difficult for VPRI to meet SLO requirements.

In contrast, VIO demonstrates remarkable stability under the same testing conditions. By decoupling IOPF handling from the critical data path, VIO avoids packet loss and retransmissions, resulting in consistent throughput that remains near 10 Gbps with minimal fluctuation. This robustness effectively mitigates I/O jitter, ensuring predictable I/O performance even under bursty or high-load conditions. Consequently, VIO enables CSPs to achieve better compliance with I/O SLOs while reducing the operational burden of jitter management, making it a reliable solution for production-scale environments.

4.4 VIO in Production

We have deployed VIO in one of the largest global CSPs for over a year. During this time, we collected extensive production data from its deployment. Throughout the upgrade and operational phases, there have been no user-reported issues related to I/O SLOs. Below, we present actual production data gathered from the deployment of VIO.

IOPF in High Oversubscription: Figure 13 illustrates the distribution of page accesses and the single IOPF observed during the trace. Under a 30% memory oversubscription rate, we analyzed a page access trace over a one-hour period using YCSB Workload on Redis.[5] During this time, the system recorded 1,464,225 unique page accesses but encountered only a single IOPF. This result highlights the effectiveness of the lockpage mechanism in maintaining low IOPF rates even under high memory pressure.

The lockpage mechanism ensures that frequently accessed I/O pages are pinned in memory, reducing the need for costly

[5]YCSB, https://github.com/brianfrankcooper/YCSB

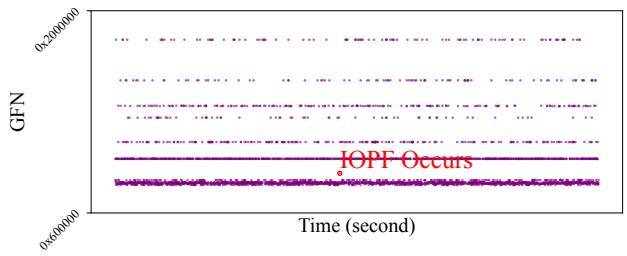


Figure 13: Page Access Trace (1:5000 Downsample). The figure shows a one-hour YCSB Workload on Redis under 30% memory oversubscription, with only one IOPF observed.

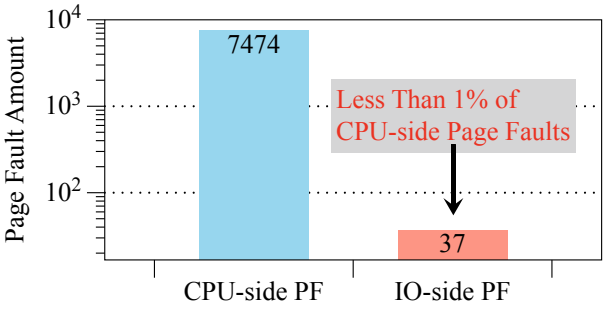


Figure 14: Daily Page Fault Comparison. The figure shows IOPFs reduced to less than 1% of CPU Page Faults, highlighting the effectiveness of the lockpage mechanism in ensuring I/O SLO requirement.

page faults. In this case, the mechanism effectively identified and locked critical pages, allowing the workload to run with minimal interruptions.

Daily Page Faults Rate: As shown in Figure 14, the lockpage mechanism in the production environment effectively reduced IOPFs to less than 1% of CPU Page Faults. By pinning frequently accessed I/O pages in memory, it minimized the occurrence of costly page faults, ensuring stable and efficient I/O operations for VMs.

This reduction in IOPFs is critical for meeting I/O SLOs. It prevents performance degradation caused by resource contention, ensuring low latency and consistent performance for critical workloads, even in high-density, multi-tenant environments.

Ablation Study: To evaluate the individual contributions of VIO's components, we conducted an ablation study using the netperf TCP_RR benchmark. [6] As shown in Figure 15, in snooping mode we observed a 3.4% performance difference between VIO without lockpage and VIO with lockpage (870,000 to 900,000 transactions per second). This improvement is attributed to the lockpage hit mechanism reducing snooping overhead by approximately 1us per operation. At

[6] netperf, https://hewlettpackard.github.io/netperf/

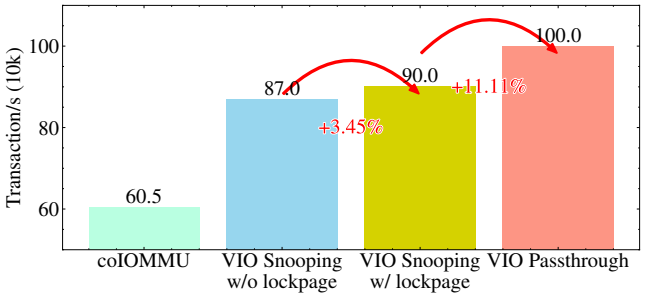


Figure 15: Ablation study of VIO under netperf TCP_RR. Lockpage improves snooping performance by 3.4%, while passthrough mode outperforms snooping by 11.1%.

high IOPS, VIO operating in snooping mode exhibits a 10% performance degradation compared to VIO in passthrough mode. This performance difference highlights the overhead of snooping under heavy I/O load and underscores the necessity for Elastic Passthrough to switch to passthrough mode in such scenarios. For comparison, we implemented colOMMU and observed that its performance at high IOPS reached only 60.5% of the native passthrough performance. This result indicates that colOMMU is less suitable for production environments with demanding I/O workloads.

5 Related Work

Static Memory Pinning Avoidance: To solve the problem of static memory pinning caused by IOPFs, PCI-SIG proposed the ATS/PRI specification [3]. However, ATS/PRI has not been widely adopted by I/O devices. Both industry and academia have explored various ways to address this issue. Previous research [17, 18, 27, 35] introduced IOPF handling into RDMA and DPU devices using methods similar to PRI from a hardware perspective. However, these PRI-like methods suffer from significant performance degradation as the number of IOPFs increases because they handle page faults in the critical path [16, 18, 49]. There have also been approaches for handling IOPF in GPUs [22, 38], but these methods are specific for GPUs, not suitable for general network and storage devices.

Many software-based solutions have also been proposed to solve the IOPF problem. coIOMMU [39] and vIOMMU [8] use para-virtualization to support IOPFs, where the guest kernel and the hypervisor work together to implement dynamic memory pinning. IOGuard [12] dedicates one CPU core to handle IOPFs in software. These software solutions either require modifications to the VM, leading to poor performance, or consume valuable CPU resources, which is unacceptable in cloud environments.

On the other hand, VIO provides a high-performance and cost-effective way to handle IOPFs without requiring changes to the guest or additional hardware features.

Memory Page Tracking: Various studies have aimed at detecting hot pages to reduce access latency by pinning or placing them in faster memory. Some approaches track memory access by utilizing page faults [23, 25, 45], while others focus on periodically checking reference bits in PTEs to monitor page hotness [26, 32, 36, 42, 48]. Modern processors provide advanced hardware features for performance monitoring, such as Intel's PEBS and AMD's IBS, which enable precise tracking of memory access events [10]. More recent works [13, 26, 47] have adopted these hardware-based techniques for enhanced memory access monitoring. However, a key challenge is balancing the trade-off between the accuracy of sampling and the overhead it incurs, especially as memory usage scales. To overcome this, a novel approach [14] combines TLB and PTE sampling, based on the observation that frequently accessed pages tend to experience a high rate of TLB misses. VIO takes advantage of its VirtIO data plane to track I/O page access in a way that is transparent to both the device and the guest OS.

6 Discussion

6.1 Compatibility

In this paper, we present a Linux-based implementation of VIO, which ensures high compatibility for several key reasons. First, VIO is built upon the VirtIO interface, a standardized virtual I/O interface that is supported across various CPU architectures and does not depend on specific hardware I/O devices or kernel versions. This makes the implementation highly portable and flexible. Second, VIO does not require any modifications to the guest OS, as it operates transparently through the VirtIO interface. This ensures that VIO maintains full compatibility with a wide range of guest OSes, including Linux, BSD, and even Windows, without enforcing any changes or additional dependencies. In theory, VIO can work with other hypervisors like Xen, Firecracker which supports VirtIO for efficient I/O virtualization. Moreover, VIO's design leverages basic virtualization features of the system, ensuring broad hardware compatibility. CPU platforms, including PowerPC, ARM, and RISC-V, can theoretically support VIO, as they widely adopt VirtIO in their virtualization ecosystems.

VIO requires guest VMs to use VirtIO and assumes the presence of a DPU that supports VirtIO offload. Major DPU vendors such as Intel and Mellanox support this functionality [5, 6], and our in-house DPU also implements it. In principle, any kernel version with VirtIO support can work with VIO. In our deployment, we have successfully migrated legacy VMs—including a 10-year-old CentOS 5 instance with kernel version 2.6.18—to VIO without modification.

6.2 Generalizability

While VIO is initially implemented and evaluated within the VirtIO framework, its underlying principles of IOPA-Snooping and Elastic Passthrough are applicable to other I/O architectures. Within the virtIO ecosystem, VIO effectively supports both network devices (virtio-net) and storage devices (virtio-blk), demonstrating its versatility. Furthermore, we have successfully extended VIO's concepts to the NVMe protocol. In NVMe, I/O submission involves writing the tail index to a doorbell register. We adapted IOPA-Snooping by intercepting this doorbell register write, enabling us to monitor I/O requests and proactively handle potential IOPFs. Additionally, we implemented Elastic Passthrough for NVMe by monitoring IOPS and dynamically switching between snooping and direct I/O, similar to the VirtIO implementation. These implementations across VirtIO-net, VirtIO-blk, and NVMe demonstrate that VIO's core ideas can be generalized beyond the VirtIO model, offering a more broadly applicable solution for managing I/O in virtualized environments.

6.3 Future Work

The VirtIO 1.1 standard draft, introduced in 2018, offers the Packed virtqueue as a more efficient alternative to the Split virtqueue [1]. Future iterations of VIO are planned to support VirtIO 1.1, further improving I/O efficiency and scalability.

7 Conclusion

This paper presents VIO, a production-level I/O virtualization framework offering hardware independence, high performance, and resource efficiency. VIO introduces: 1) an elastic passthrough mechanism adapting to IOPS demands for near-native performance; 2) an adaptive lockpage mechanism reducing IOPFs by pinning frequently accessed pages; and 3) a proactive snooping approach separating IOPF handling from the critical I/O path. Unlike PRI, which struggles with latency and limited hardware adoption, VIO provides a practical and efficient solution for large-scale cloud environments.

VIO, deployed across 300K VMs, achieves up to the equivalent of 30K VM memory daily savings while maintaining I/O SLOs with less than 1% IOPFs compared to CPU page faults, demonstrating its efficiency and scalability in production environments.

8 Acknowledgements

We thank our shepherd and the anonymous reviewers for the insightful comments that improved the quality of this paper. This work was supported in part by the National NSF of China (NO. 62141218, 62232012), and Shanghai Key Laboratory of Scalable Computing and Systems.

References

[1] Virtual I/O Device (VIRTIO) Version 1.1. https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html, Dec. 2018. [Online; accessed 14. May 2025].

[2] mm / virtio: Provide support for free page reporting. https://lwn.net/Articles/808807, Jan. 2020. [Online; accessed 9. May 2025].

[3] Address Translation Services Revision 1.1 | PCI-SIG, Dec. 2024. [Online; accessed 6. Dec. 2024].

[4] Linux Plumbers Conference 2024. https://lpc.events/event/18/contributions/1684, May 2025. [Online; accessed 14. May 2025].

[5] Powering Infrastructure to Help Shape the Data Center of the Future. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-09/ipu-baidu-dpu-white-paper.pdf, May 2025. [Online; accessed 15. May 2025].

[6] VirtIO-net Emulated Devices - NVIDIA Docs. https://docs.nvidia.com/networking/display/bluefielddpubspv422/virtio-net+emulated+devices, May 2025. [Online; accessed 15. May 2025].

[7] ALLEN, T., AND GE, R. In-depth analyses of unified virtual memory system for gpu accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2021), SC '21, Association for Computing Machinery.

[8] AMIT, N., BEN-YEHUDA, M., TSAFRIR, D., AND SCHUSTER, A. viommu: Efficient IOMMU emulation. In *Proceedings of the 2011 USENIX Annual Technical Conference, USENIX ATC 2011, Portland, OR, USA, June 15-17, 2011* (2011), J. Nieh and C. A. Waldspurger, Eds., USENIX Association.

[9] AMIT, N., AND WEI, M. The design and implementation of hyperupcalls. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. C. Reed, Eds., USENIX Association, pp. 97–112.

[10] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings* (2010), R. H. Arpaci-Dusseau and B. Chen, Eds., USENIX Association, pp. 423–436.

[11] CHEN, Y., BILAS, A., DAMIANAKIS, S. N., DUBNICKI, C., AND LI, K. Utlb: a mechanism for address translation on network interfaces. *SIGOPS Oper. Syst. Rev. 32*, 5 (Oct. 1998), 193–204.

[12] DONG, Y., AND MI, Z. Ioguard: Software-based I/O page fault handling with one CPU core. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware, Internetware 2024, Macau, SAR, China, July 24-26, 2024* (2024), H. Mei, J. Lv, A. Helal, X. Ma, S. Cheung, J. Zhang, and T. Zhang, Eds., ACM.

[13] DURAISAMY, P., XU, W., HARE, S., RAJWAR, R., CULLER, D. E., XU, Z., FAN, J., KENNELLY, C., MCCLOSKEY, B., MIJAILOVIC, D., MORRIS, B., MUKHERJEE, C., REN, J., THELEN, G., TURNER, P., VILLAVIEJA, C., RANGANATHAN, P., AND VAHDAT, A. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023* (2023), T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., ACM, pp. 727–741.

[14] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A. M., CHUNG, E. S., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. G. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (2018), S. Banerjee and S. Seshan, Eds., USENIX Association, pp. 51–66.

[15] FUKUOKA, T., SATO, S., AND TAURA, K. Pitfalls of infiniband with on-demand paging. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2021), pp. 265–275.

[16] FUKUOKA, T., SATO, S., AND TAURA, K. Pitfalls of infiniband with on-demand paging. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021* (2021), IEEE, pp. 265–275.

[17] GUO, K., LI, D., LUO, B., SHEN, Y., PENG, K., LUO, N., DAI, S., LIANG, C., SONG, J., YANG, H., ZHANG, X., AND MI, Z. VPRI: efficient I/O page fault handling via software-hardware co-design for iaas clouds. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024* (2024), E. Witchel, C. J. Rossbach, A. C. Arpaci-Dusseau, and K. Keeton, Eds., ACM, pp. 541–557.

[18] GUO, Z., SHAN, Y., LUO, X., HUANG, Y., AND ZHANG, Y. Clio: a hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 417–433.

[19] HAR'EL, N., GORDON, A., LANDAU, A., BEN-YEHUDA, M., TRAEGER, A., AND LADELSKY, R. Efficient and scalable paravirtual I/O system. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 231–242.

[20] HUDZIA, B. On allowing shorter timeout on mellanox cards and other tips and tricks. https://www.reflectionsofthevoid.com/2014/02/on-allowing-shorter-timeout-on-mellanox.html, 2014. Accessed: 2024-12-10.

[21] JALEEL, A., EBRAHIMI, E., AND DUNCAN, S. Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems. *ACM Trans. Archit. Code Optim. 16*, 1 (Mar. 2019).

[22] KIM, H., SIM, J., GERA, P., HADIDI, R., AND KIM, H. Batch-aware unified memory management in gpus for irregular workloads. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 1357–1370.

[23] KIM, J., CHOE, W., AND AHN, J. Exploring the design space of page management for multi-tiered memory systems. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 715–728.

[24] KUPERMAN, Y., MOSCOVICI, E., NIDER, J., LADELSKY, R., GORDON, A., AND TSAFRIR, D. Paravirtual remote i/o. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, Association for Computing Machinery, p. 49–65.

[25] LAGAR-CAVILLA, H. A., AHN, J., SOUHLAL, S., AGARWAL, N., BURNY, R., BUTT, S., CHANG, J., CHAUGULE, A., DENG, N., SHAHID, J., THELEN, G., YURTSEVER, K. A., ZHAO, Y., AND RANGANATHAN, P. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019* (2019), I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, pp. 317–330.

[26] LEE, T., MONGA, S. K., MIN, C., AND EOM, Y. I. MEMTIS: efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 17–34.

[27] LESOKHIN, I., ERAN, H., RAINDEL, S., SHAPIRO, G., GRIMBERG, S., LISS, L., BEN-YEHUDA, M., AMIT, N., AND TSAFRIR, D. Page fault support for network controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017* (2017), Y. Chen, O. Temam, and J. Carter, Eds., ACM, pp. 449–466.

[28] LESOKHIN, I., ERAN, H., RAINDEL, S., SHAPIRO, G., GRIMBERG, S., LISS, L., BEN-YEHUDA, M., AMIT, N., AND TSAFRIR, D. Page fault support for network controllers. *SIGARCH Comput. Archit. News 45*, 1 (Apr. 2017), 449–466.

[29] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High performance VMM-Bypass I/O in virtual machines. In *2006 USENIX Annual Technical Conference (USENIX ATC 06)* (Boston, MA, May 2006), USENIX Association.

[30] LIU, Y., MA, J., ZHANG, Z., LI, L., QI, Z., AND GUAN, H. Megatron: Software-managed device tlb for shared-memory fpga virtualization. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (2021), pp. 1213–1218.

[31] PARK, J., JEONG, D., AND KIM, J. Uvmmu: Hardware-offloaded page migration for heterogeneous computing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2023), pp. 1–6.

[32] PARK, S., BHOWMIK, M., AND UTA, A. DAOS: data access-aware operating system. In *HPDC '22: The 31st International Symposium on High-Performance Parallel and Distributed Computing, Minneapolis, MN, USA, 27 June 2022 - 1 July 2022* (2022), J. B. Weissman, A. Chandra, A. Gavrilovska, and D. Tiwari, Eds., ACM, pp. 4–15.

[33] PCI-SIG. Address translation services revision 1.1, 2009. Accessed: 2024-12-10.

[34] PICHAI, B., HSU, L., AND BHATTACHARJEE, A. Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces. *SIGPLAN Not. 49*, 4 (Feb. 2014), 743–758.

[35] PSISTAKIS, A., CHRYSOS, N., CHAIX, F., ASIMINAKIS, M., GIANNIOUDIS, M., XIROUCHAKIS, P., PAPAEFSTATHIOU, V., AND KATEVENIS, M. PART: pinning avoidance in RDMA technologies. In *14th IEEE/ACM International Symposium on Networks-on-Chip, NOCS 2020, Hamburg, Germany, September 24-25, 2020* (2020), IEEE, pp. 1–8.

[36] REN, J., XU, D., RYU, J., SHIN, K., KIM, D., AND LI, D. MTM: rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024* (2024), ACM, pp. 803–817.

[37] STEVE SISTARE, M. K. QEMU Live Update. https://blogs.oracle.com/linux/post/qemu-live-update, May 2025. [Online; accessed 14. May 2025].

[38] TANASIC, I., GELADO, I., JORDÀ, M., AYGUADÉ, E., AND NAVARRO, N. Efficient exception handling support for gpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017* (2017), H. C. Hunter, J. Moreno, J. S. Emer, and D. Sánchez, Eds., ACM, pp. 109–122.

[39] TIAN, K., ZHANG, Y., KANG, L., ZHAO, Y., AND DONG, Y. coiommu: A virtual IOMMU with cooperative DMA buffer tracking for efficient memory management in direct I/O. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 479–492.

[40] VOGEL, P., MARONGIU, A., AND BENINI, L. Exploring shared virtual memory for fpga accelerators with a configurable iommu. *IEEE Transactions on Computers 68*, 4 (2019), 510–525.

[41] WALDSPURGER, C. A. Memory resource management in vmware ESX server. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002* (2002), D. E. Culler and P. Druschel, Eds., USENIX Association.

[42] WANG, Y., CHEN, L., DENG, T., LUO, B., SHEN, Y., WEI, Z., XU, Y., HUANG, M., AND QI, Z. Effectively virtual page prefetching via spatial-temporal patterns for memory-intensive cloud applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2025)* (2025). To appear.

[43] WANG, Y., LUO, B., AND SHEN, Y. Efficient memory overcommitment for I/O passthrough enabled vms via fine-grained page meta-data management. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023* (2023), J. Lawall and D. Williams, Eds., USENIX Association, pp. 769–783.

[44] WEINER, J., AGARWAL, N., SCHATZBERG, D., YANG, L., WANG, H., SANOUILLET, B., SHARMA, B., HEO, T., JAIN, M., TANG, C., AND SKARLATOS, D. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 609–621.

[45] WEINER, J., AGARWAL, N., SCHATZBERG, D., YANG, L., WANG, H., SANOUILLET, B., SHARMA, B., HEO, T., JAIN, M., TANG, C., AND SKARLATOS, D. TMO: transparent memory offloading in datacenters. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 609–621.

[46] WÜNSCHE, J., KARIM, S., KUHN, M., BRONESKE, D., AND SAAKE, G. Intelligent data migration policies in a write-optimized copy-on-write tiered storage stack. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (New York, NY, USA, 2023), CHEOPS '23, Association for Computing Machinery, p. 17–26.

[47] XIANG, L., LIN, Z., DENG, W., LU, H., RAO, J., YUAN, Y., AND WANG, R. Nomad: Non-exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 19–35.

[48] YAN, Z., LUSTIG, D., NELLANS, D. W., AND BHATTACHARJEE, A. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019* (2019), I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, pp. 331–345.

[49] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Filemr: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (2020), R. Bhagwan and G. Porter, Eds., USENIX Association, pp. 111–125.

[50] ZHANG, X., ZHENG, X., WANG, Z., LI, Q., FU, J., ZHANG, Y., AND SHEN, Y. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019* (2019), I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, pp. 93–105.

Enabling Efficient GPU Communication over Multiple NICs with FuseLink

Zhenghang Ren[1] Yuxuan Li[1] Zilong Wang[1] Xinyang Huang[1] Wenxue Li[1] Kaiqiang Xu[1]
Xudong Liao[1] Yijun Sun[1] Bowen Liu[1] Han Tian[2] Junxue Zhang[1] Mingfei Wang[3]
Zhizhen Zhong[4] Guyue Liu[5] Ying Zhang[6] Kai Chen[1]

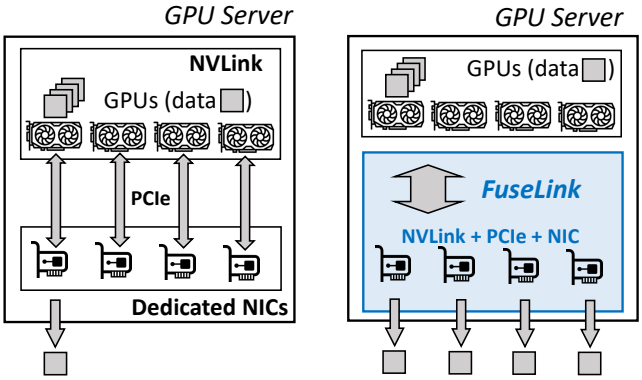
[1]iSINGLab, Hong Kong University of Science and Technology
[2]University of Science and Technology of China [3]MetaX Integrated Circuits
[4]Massachusetts Institute of Technology [5]Peking University [6]Meta

Abstract

Machine learning (ML) clusters stack multiple network interface cards (NICs) within each server to improve inter-server GPU communication bandwidth. However, existing systems fall short in fully utilizing NICs because of static GPU-NIC bindings. This leads to bottlenecks at hot-spot NICs when handling imbalanced communication in ML tasks. For example, large language model serving instances may have different communication demands across NICs; expert-parallel training tasks have imbalanced all-to-all traffic; and the embedding transmission volumes during recommendation model training vary across GPUs. To fully utilize all NICs, we propose FuseLink to enable efficient GPU communication over multiple NICs. FuseLink extends inter-server network by integrating high-speed intra-server connections, and leverages GPUs to efficiently relay traffic to idle NICs. We implement FuseLink and integrate it into NCCL, so that ML applications can benefit from FuseLink seamlessly without code modifications. Compared to NCCL, we demonstrate that FuseLink achieves up to 212GBps bandwidth between two inter-server GPUs and accelerates ML tasks with dynamic traffic patterns. Specifically, it reduces the latencies of first-token generation in LLM model servings by 1.04-2.73×, improves the training throughput of mixture-of-experts model by up to 1.3×, and accelerates deep learning recommendation model training by up to 1.2×.

1 Introduction

Efficient communication among GPUs is crucial in distributed ML tasks, especially given the expanding scale of AI infrastructure [9, 27, 32, 35, 53, 68]. Distributed ML tasks are typically bounded by GPU communication bandwidth [26, 32, 48, 54], driving the speed increase of both intra-server [1] and inter-server network. For example, NVLink [14, 39, 46, 52, 57] connects neighbouring GPUs with Tbps bandwidth; RDMA [21, 49, 66] offers inter-server connections with hundreds of Gbps bandwidth. However, RDMA NIC bandwidth grows slower compared to intra-server bandwidth [11], leaving inter-server connection a major bottleneck in distributed ML tasks.



(a) Static GPU-NIC binding (b) FuseLink

Figure 1: Current static GPU-NIC binding and FuseLink

To mitigate inter-server bottleneck, existing GPU clusters stack multiple NICs within each server to improve inter-server bandwidth [2, 23]. Currently, they bind each GPU communication to a specific PCIe-based NIC, as shown in Figure 1a. Note that the figure omits the hierarchical structure of PCIe [7] which leads to non-uniform bandwidths when a NIC access GPUs through different paths. There usually exists only one NIC that can provide full bandwidth for each GPU during inter-server communication, while other NIC bandwidths are limited by suboptimal PCIe paths.

Static binding is suitable for ML tasks with balanced traffic. For instance, 3D parallel training [42, 53] divides models into equal shards, producing identical traffic pattern across GPU workers [34]. However, for widely-deployed ML tasks that exhibit *imbalanced* inter-server traffic across NICs, stacking NICs with static traffic binding does not necessarily improve inter-server communication efficiency. Some workers may

[1]Some GPU connections, like NVLink, link GPUs within a short range between servers. However, we will still call them "intra-server connections" to differentiate them from "inter-server connections" that use NICs.

demand higher bandwidth than a single NIC (§2.1). Examples include disaggregated LLM serving [29, 40, 47, 69], mixture-of-experts (MoE) models [18, 55], and deep learning recommendation models (DLRM) [10, 25]. This traffic imbalance can lead to suboptimal performance, with some NICs being bottlenecks while other NICs remain idle. Moreover, the causes of traffic imbalance vary significantly, such as unpredictable input requests, model architectures, and parallelism methods, making it difficult to achieve optimal inter-server communication through static NIC load balancing strategies.

To fill this gap, we ponder a fundamental question: *can arbitrary ML workloads effectively leverage all available RDMA NICs for data transmission, aggregating links over multiple NICs as a "FuseLink" for inter-server communication?* If so, we could significantly improve GPU communication bandwidth, surpassing the limitations of static NIC binding, and achieve optimal NIC utilization in dynamic-traffic ML tasks, without requiring adjustments to parallelism strategies or modifications to application code.

We build FuseLink, a GPU communication framework that achieves optimal NIC utilization for distributed machine learning by enabling GPUs to flexibly transmit data through multiple NICs. Figure 1a shows a typical GPU server equipped with multiple GPUs and RDMA NICs, with each RDMA NIC statically bound to one GPU through PCIe switches (bridges). In contrast, Figure 1b illustrates FuseLink, which disaggregates the network resources of the inter-server network and enables flexible NIC utilization.

FuseLink's key idea is to integrate high-speed intra-server links as critical extensions of the inter-server network. Existing GPU communication frameworks, such as NCCL [45], leverages NVLink to improve throughput on NICs with suboptimal PCIe paths. However, these methods remains static to a specific NIC, lacking of abilities to dynamically balance NIC workloads or aggregating bandwidths. This is inherently limited by the incompatibility of intra- and inter-server networks, preventing access through NICs and NVLinks directly.

Our insight is that intra-server network can be seamlessly integrated to inter-server network by scheduling NIC workload and efficiently configure traffic relaying via intra-server connections. Additionally, we exploit characteristics of ML applications that they typically have a limited numbers of inter-server connections and send large messages in chunks [26, 32, 41, 42, 54]. These characteristics allows for efficient scheduling of traffic across NICs with runtime intra-server traffic relaying. As a result, FuseLink effectively leverages multiple NICs, aggregating multi-NIC bandwidths and significantly improving overall NIC utilization.

We build FuseLink on Nvidia platform with the following design goals:

Maximizing multi-NIC efficiency. When transmitting data from a GPU through multiple NICs, a key challenge is that multi-NIC transmission is bounded by PCIe bandwidth and PCIe topology that lacks direct connections between the GPU and NICs. We refer these NICs as *indirect NICs*. FuseLink enables efficient multi-NIC transmission by exploiting NVLink for data relaying on both sender and receiver sides.

Avoiding contention and interruption. Utilizing relay GPUs and multiple NICs should not interrupt or slow down ongoing computation and communication tasks by consuming extra memory and bandwidth resources. FuseLink avoids such interruption by employing priority-based memory management and network requests scheduling.

Readily deployable. FuseLink is designed to work with existing hardware and ML frameworks commonly found in GPU clusters, which allow benefiting from FuseLink without modifying ML applications. We implement FuseLink on top of RDMA primitives as an independent networking layer to replace the default networking system, and integrate FuseLink into NCCL [45], allowing to use FuseLink seamlessly.

We evaluate FuseLink on Nvidia GPUs with eight-lane NVLinks and eight 400Gbps NICs for inter-server communication. Our evaluation shows that FuseLink significantly improves inter-server GPU communication bandwidth, achieving 212GBps between two inter-server GPUs, surpassing the eight-lane NVLinks bandwidth. End-to-end evaluations show that FuseLink effectively accelerates ML tasks with traffic imbalance, including LLM serving task time-to-first-token (TTFT [2]) by 1.04-2.73×, MoE training by 1.3×, and DLRM training by 1.2×.

This paper makes the following key contributions:

- We design FuseLink that boosts inter-server GPU communication over multiple NICs by integrating high-speed intra-server GPU connections (e.g., NVLinks).
- We integrate FuseLink into NCCL, enabling ML applications to benefit from it seamlessly.
- FuseLink effectively improves inter-server GPU bandwidth and ML tasks efficiency under imbalanced traffic.

2 Background

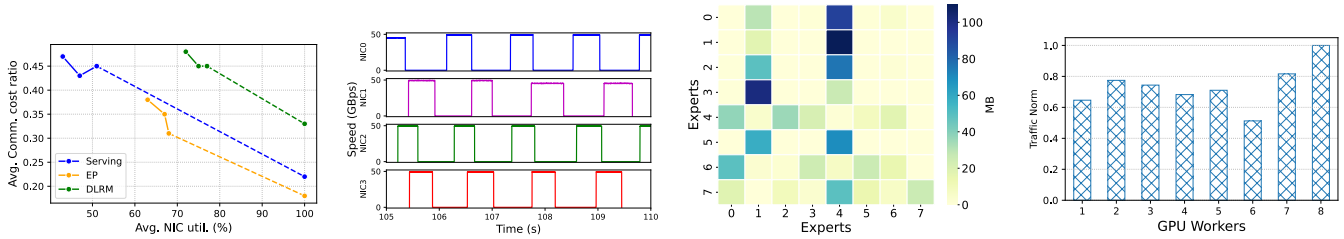
High-speed interconnect is essential to facilitate large scale machine learning. With the growing model sizes and expanding dataset volumes [16, 59, 60], real-world ML applications have been typically deployed in distributed environments [54, 56, 68], incurring large communication overhead and thus demanding high-bandwidth networks.

There have been substantial efforts on improving GPU communication bandwidth, including dedicated intra-server GPU connections and inter-server network. Industrial practices have been developing fast dedicated GPU interconnect as an integral part in ML clusters, providing Tbps network

[2] Inter-server communication impacts either time-to-the-first token (TTFT) or time-to-the-second-token (TTST), depending on when the prompt cache is transmitted to the decode phase. Here we transmit the cache before generating the first token, which affects TTFT.

ML Tasks	Communication Pattern | Cause			NIC util.	Comm. ratio
	via direct NICs	Concurrent transmission	Same traffic volume		
Distributed LLM Serving	✓	✗ | stochastic task arrival	✗ | varying requests sizes	13%-53%	11%-82%
Expert-parallel MoE	✓	✗ | async. transmission	✗ | different token batches	29%-65%	15%-42%
DLRM	✓	✓	✗ | different embeddings	59%-82%	28%-55%

Table 1: Dynamic-traffic ML tasks with communication patterns, ✓(✗) marks satisfied (unsatisfied) patterns



(a) Avg. NIC util. and comm. cost (b) Model serving (c) Expert parallelism (d) DLRM

Figure 2: Traffic imbalance and communication cost in ML tasks

between GPUs. For example, Nvidia builds NVLink and NVSwitch [46] to connect near-range GPUs; Google designs dedicated inter-chip-interconnect (ICI) for TPU groups [20]. Inter-server network [27], however, has relatively lower bandwidth on a single NIC than intra-server network. Therefore, GPU clusters typically stack multiple NICs within each server to satisfy inter-server bandwidth demand of GPUs [43]. The common configuration of NICs is to have the same number of NICs as GPUs within a server and install each NIC on the PCIe slot with direct access to the corresponding GPU slot. This ensures each GPU can get the full bandwidth of its NIC.

Despite the efforts of improving inter-server bandwidth, existing systems fail to fully utilize inter-server bandwidth for ML tasks with dynamic traffic because of the static binding of GPU traffic and NICs that causes low NIC utilization. The dynamic traffic is intrinsic to widely adopted ML tasks, including model serving [33, 37, 69], MoE models [15, 18, 55], and embedding transmission in recommendation models [10, 25]. Moreover, the NIC load imbalance is more pronounced across large GPU interconnect domains such as the NVL72 [3] system, where 72 GPUs and NICs are connected with NVLink and RDMA. We analyze the root cause by investigating ML tasks with dynamic communication patterns.

2.1 Deficiency in Dynamic-traffic ML

We test typical ML tasks with dynamic traffic shown in Table 1. Our tests are conducted on servers each with eight Hopper GPUs connected via eight-lane NVLink and eight 400Gbps NICs for inter-server network. GPU traffic is transmitted via direct NICs to maximize NIC throughput. Our metrics are NIC utilizations and the ratio of communication in ML tasks. We estimate NIC utilizations by the speed gap between static NIC binding and the ideal situation where GPUs can use all available NICs for communication. Figure 2a shows the average NIC utilizations and communication ratios in total costs, together with the estimated ideal utilization where we exploit all NICs to accelerate inter-server data transmission.

Disaggregated LLM serving. We test typical disaggregated LLM serving cases where model serving is split into prefill phase and decode phase [47, 50, 69], with processed requests sent from prefill phase to decode phase. We feed the input requests from public traces [5] randomly and record NIC utilizations when transmitting requests to decode stages. The results shows only 13%-53% of NIC utilization when transmitting data between phases, because serving traffic is stochastic, depending on task arrival times and request lengths, with traffic volumes ranging from hundreds of MBs to GBs. Even with advanced optimization tricks that overlap communication and computation [47, 50], we cannot fully utilize NICs because requests vary across GPU workers. Figure 2b depicts the recorded four NIC speeds during model serving, which shows idleness of some NICs while other NICs are busy.

MoE training. We test MoE model [18, 55] training on Mixtral 8×7B model with expert parallelism (EP) degree of eight and tensor parallel degree of four. We record NIC utilizations when GPU workers transmit tokens to peer workers with the activated experts. NIC utilization shows 29%-65%, with communication taking 28%-55% of the total cost. The root causes are asynchronized inter-server GPU communication and different token batches of experts during MoE training. Figure 2c shows the traffic map between expert models. EP training allocates expert models to GPU workers, with each worker training a subset of experts. The imbalanced traffic persists even with equal model partitions because expert layers are sparsely activated for each input token based on the output of gate layers. Since data samples are routed to different experts, GPU workers need to dispatch data to and collect

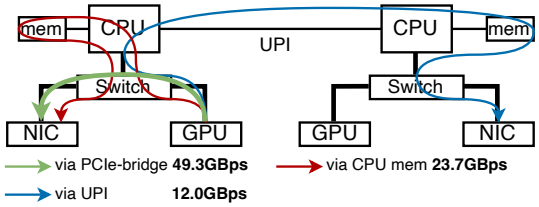


Figure 3: Conceptual intra-server PCIe topology with NIC bandwidth on different PCIe paths

data from peer workers, which leads to imbalanced all-to-all traffic across NICs. With different traffic volumes among experts, static binding of NICs and GPUs fails to fully utilize NICs bandwidth when all-to-all communication is bounded by the GPU pair with the largest traffic volume.

DLRM We test DeepFM [22], a deep learning based recommendation model with Avazu [61] dataset. DLRM [10, 25] consists of dense models and large lookup tables that store semantic embedding vectors of objects. In each training iteration, GPU workers fetch embeddings and push gradients from/to the server with the embedding table if they are not cached locally and train dense models with data parallelism. Embedding transmission volumes vary across workers, leading to different traffic volumes. Our results show that embedding transmissions achieve 59%-82% NIC utilization, and take up to 55% of the total cost. Figure 2d shows the imbalanced traffic distribution among eight GPU workers which causes performance to be bounded by the GPU worker with the largest embedding transmission.

2.2 Opportunities

Leveraging idle NICs. By analyzing dynamic-traffic ML tasks in Figure 2, we conclude that static NIC binding is not well-suited for such tasks. This is because achieving optimal NIC utilization with static NIC binding requires *transmission through direct NICs, concurrent transmission, and equal traffic volumes*. Without these conditions, communication performance is limited by tasks bound to specific NICs, while other NICs remain idle. This idle NIC capacity can instead be exploited to accelerate communication by dynamically scheduling traffic to idle NICs.

Multi-NIC communication is non-trivial because it does not improve inter-server bandwidth with only PCIe connections. On one hand, the total bandwidth is limited when all NICs access through the single PCIe interface of sender GPU. On the other hand, some NICs are indirectly connected with the sender GPU, with data traversing through PCIe root complex or even across NUMA, leading to suboptimal NIC throughput. With NVLink and imbalanced traffic in ML tasks, we can significantly improve efficiency on multi-NIC transmission.

Dynamic NVLink routing to bypass PCIe. Existing GPU communication frameworks, such as NCCL [45], leverage NVLink to bypass suboptimal PCIe paths when transmitting data through indirect NICs. However, they focus only on optimizing a single NIC, leaving other NICs restricted to suboptimal PCIe paths. This limitation prevents efficient utilization of multiple NICs for high-performance data transmission. Figure 3 presents the NIC speeds across different paths. The bandwidth of inter-server GPU communication is constrained by the PCIe interface and topology, particularly when using indirect NICs [30], which require data to traverse suboptimal paths. To address this, we enable dynamic traffic routing to peer GPUs via NVLink, allowing the efficient aggregation of multiple NICs' bandwidth for inter-server GPU communication.

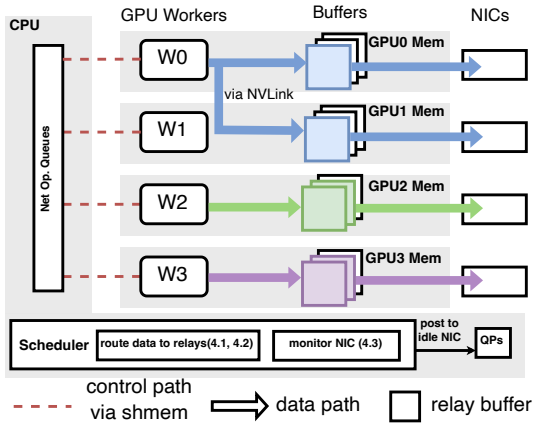


Figure 4: **FuseLink architecture in a GPU server** with four parallel GPU workers and four NICs. W0 is using both NIC 0 and NIC 1 for inter-server communication when W1 is not sending any traffic. W0 sending buffers are mapped to GPU0 and GPU1 for efficient data transmission

3 FuseLink Overview

FuseLink achieves multi-NIC communication by scheduling worker's traffic to multiple NICs dynamically with awareness of NIC load status, instead of statically binding the traffic to specific NICs. To effectively leverage indirect NICs, FuseLink routes traffic via intra-server connection to optimal GPUs for NIC access. The primary design goal of FuseLink is to achieve efficient multi-NIC transmission while minimizing contention risks, and is compatible with existing ML infrastructures. Figure 4 shows FuseLink architecture on sender side with an example workflow.

FuseLink initializes with inter-server connection setup. FuseLink explores the intra-server topology and establishes RDMA connections. During connection setup, FuseLink identifies available NICs for inter-server communication. Based on the topology, FuseLink selects the optimal data path for each NIC and GPU. If a NIC is directly connected to the GPU via a PCIe bridge or PCIe switch, it uses this direct path, as the NIC can access the GPU with full bandwidth. Otherwise,

FuseLink selects a router GPU for relaying data to the NIC. In Figure 4, GPU1 is the router when GPU0 sends via NIC1.

Inter-server GPU communication starts with receivers issuing *credits*, which are data structures containing necessary resources for initiating RDMA. These resources include receiving memory regions and requested data sizes. Crucially, the credits contain idle NICs information, derived from FuseLink actively monitoring NIC receiving load status. This allows senders to select the suitable NIC for sending data. Credits are common in RDMA-enabled applications [17, 28, 45, 62] because of its bypassing feature.

On sender side, FuseLink intercepts sending requests and waits for credits from receivers. At the same time, FuseLink monitors NIC sending load status. When credits arrive, FuseLink considers both sending NIC load status and receiving load status to select optimal NICs for data transmission. Moreover, FuseLink leverages intra-server GPU connection to route traffic efficiently to NICs. For instance, Figure 4 shows data routing through NVLink from GPU0 to GPU1 and sent through NIC1.

FuseLink gets notified when data arrives on receivers. Note that the data may be transmitted to a router GPU rather than the receiver GPU, because the sender may select a receiver NIC that has an indirect connection to receiver GPU. In this case, we stage receiving data on a router GPU first to ensure transmission efficiency. It then routes traffic via intra-server connection to receiver GPUs if needed.

Compared to existing GPU communication frameworks, FuseLink introduces several key advancements.

- **Disaggregated network resources.** Instead of operating on RDMA connections directly, FuseLink introduces an abstraction layer that unifies both intra-server and inter-server connection. This design enables efficient and flexible NIC utilization.

- **Dynamic NIC utilization.** FuseLink monitors NIC status in real-time rather than statically bind traffic to specific NICs. This approach allows to detect idle NICs and dynamically assign traffic to them, significantly improving overall bandwidth utilization.

- **Traffic routing with high-bandwidth GPU links.** Instead of statically binding traffic to NICs or intermediate GPUs [1, 44, 45], FuseLink fully leverage high-bandwidth GPU links for traffic routing, thereby bypassing PCIe constrains in traditional interconnects and enhancing communication efficiency.

3.1 Challenges

FuseLink supports multi-NIC transmission by allowing inter-server GPUs to dynamically schedule traffic to NICs. However, there presents several technical challenges:

Design	Inter-server BW(GBps)	Speedup
Baseline	49.27	1.0
Efficient relaying §4.1	78.39	1.59
Eliminate Interruption §4.2	76.37	1.55
Reduce NIC contention §4.3	178.59	3.62
Scheduling Efficiently §4.4	212.35	4.31

Table 2: Designs and achieved speedup over baseline

Relaying overhead. (§4.1) FuseLink introduces flexible intra-server traffic routing across GPUs to address the bandwidth limitations when transmitting through multiple NICs. However, GPUs are not inherently optimized for relaying data to NICs. Instead, both GPUs and intra-server connections are optimized for memory I/O between devices [24, 30], which is incompatible with PCIe-based NICs. This gap enforces frequent device synchronizations during network transmission, significantly slowing down the data path. For instance, when a GPU acts as a relay to transfer data from other GPUs to NICs, the data must be fully available on the relay GPU before NICs can access it. This results in frequent synchronizations between the relay GPU and sender GPUs. Consequently, the NIC throughput has minor improvements compared to the throughput in Figure 3. We address this challenge by actively planning relay data paths by memory remapping, thereby efficiently relaying data via GPUs for multi-NIC transmission.

Interruption and contention risks. (§4.2,§4.3) Maintaining performance isolation is challenging when utilizing NICs and GPU memory. These resources may contend for NIC bandwidth during data transmission, potentially disrupting tasks by draining GPU memory. Consequently, inter-server bandwidth is limited because of NIC contention, as listed in Table 2. The primary cause of the contention is FuseLink's strategy to schedule traffic to idle NICs when peer workers are not engaged in communication tasks. However, NIC idleness are temporal in dynamic-traffic ML tasks [10, 18, 47]. When peer workers initiate new communications, outstanding traffic may still in transmission, scheduled by FuseLink during periods idleness. We address this challenge by prioritizing ongoing tasks on relay GPUs and monitoring network resource usage with priority-based scheduling. This approach enables senders to identify idle NICs and eliminate interruption risks.

Scheduling overhead. (§4.4) Dynamically scheduling traffic to NICs is inherently complex, considering requirements to mitigate contention risks and to avoid traffic interference and interruption, making the reduction of control overhead a significant challenge. Existing works on communication scheduling focus mainly on task-level scheduling [51, 64, 65] to maximize network resource utilization. However, they fail to identify idle NIC resources and schedule communication efficiently. We address this issue by implementing an efficient traffic monitoring and scheduling strategy that balances monitoring accuracy with performance.

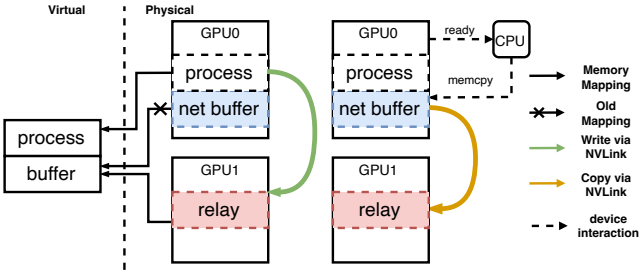


Figure 5: Candidate intra-server traffic redirection methods *D1* (left) and *D2* (right) using two GPUs as an example. The network buffer is allocated on GPU0. *D1* remaps network buffer to relay GPU and GPU threads fill relay buffer directly. *D2* initiates asynchronized memory copy between GPUs from CPU side after network buffer is ready

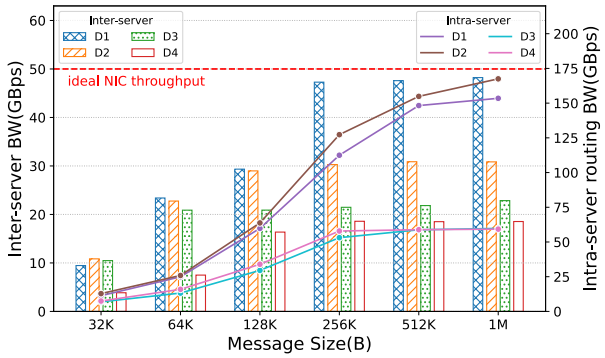


Figure 6: Intra-server relay bandwidth and indirect NIC throughput. *D1* has less intra-server bandwidth than *D2*, but has the highest indirect NIC throughput

4 FuseLink Design

We illustrate FuseLink design with a focus on solving the aforementioned challenges, achieving high NIC utilization while without interrupting peer GPUs.

4.1 Efficient Intra-server Relaying

Efficiently relaying data across GPUs to get high bandwidth on indirect NICs is difficult mainly because GPU interconnect and inter-server network are incompatible. This incompatibility enforces memory copy and device synchronizations before sending with NICs, which stalls network transmission and prolongs message latency.

To address this challenge, our insight is to exploit the existing memory I/O on network buffers during inter-server communication and redirect the traffic via memory remapping. FuseLink achieves efficient intra-server traffic relaying without modifying existing ML frameworks by taking advantages of the existing virtual address systems [6]. The system manages GPU memories in a unified virtual memory address space, which allows to map virtual memory addresses to arbitrary physical memories. With this feature, FuseLink decouples network buffers with the physical memories and remaps them to buffers on relay GPUs, so that traffic is redirected through intra-server connections via memory I/O when applications fill the network buffers.

To show the rationale of FuseLink relaying with remapping, we explore the design space by inspecting inter-server communication in two steps. ① The GPU worker fills network buffer, and ② The GPU notifies CPU to initiate RDMA transmission. To configure the data path without code modification, we have the following candidate solutions. *D1*: modify step ① so that the GPU writes data to remapped buffers on relay GPUs. *D2*: modify step ② so that the CPU copies data to relay GPUs then RDMA. *D3*: GPU threads write data to buffers mapped to host memory. *D4*: CPU initiates memory copy to host memory.

Among four candidate relaying methods, *D3* and *D4* achieve the same bandwidth as PCIe shown in Figure 3, thus limited by PCIe speed and cannot improve relaying efficiency. *D1* and *D2*, however, have pros and cons and cannot be easily compared. As shown in Figure 5. *D2* has high intra-server bandwidth when performing memory copy in a batch asynchronously, while *D1* enforces synchronized memory load and store and involves memory remapping with extra overhead. However, *D1* has only one copy step to relay GPUs, achieving higher data path efficiency and lower message latency. To this end, we set up performance benchmark to select the optimal intra-server relay method.

Benchmark indirect NIC throughput. We test GPU communication bandwidth through indirect NICs on servers with multiple GPUs, eight-lane NVLinks, and 400Gbps NICs. The theoretical bandwidth of eight-lane NVLinks is 200 GB/s. The experiment is conducted by sending messages constantly between two inter-server GPUs via indirect NICs. The traffic is routed from the sender GPU to indirect NICs with relaying methods *D1*-*D4* respectively. We record intra-server traffic relaying bandwidth and throughput on the indirect NIC under different data chunk sizes. Figure 6 shows the intra-server relaying bandwidth and indirect NIC throughput.

Among four relaying methods, *D1* achieves the highest throughput on the indirect NIC, compared to *D2*-*D4*. *D3* and *D4* route data to host memory via PCIe, thus are limited by PCIe bandwidth. In contrast, *D1* and *D2* exploit NVLink connections, achieving much higher relaying bandwidth. When comparing *D1* and *D2*, we observed that *D2* has higher intra-server relaying throughput, because CPU-initiated memory copy between devices batches memory copy, which achieves higher throughput than GPU-initiated memory copy.

However, *D1* achieves the highest NIC throughput. The reason is that *D1* has the most efficient data path between the GPU and indirect NICs through memory remapping, which brings the following benefits:

① High indirect NIC throughput: *D1* remaps network buffer to peer GPUs, so that relaying is done when GPUs fill send-

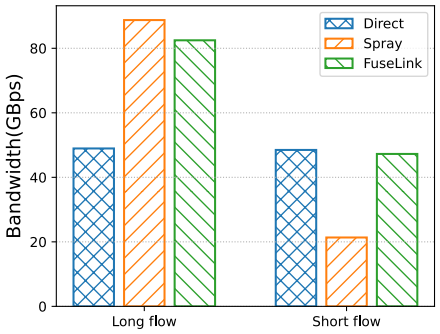


Figure 7: Performance of long flow and short flow under direct NIC communication, traffic spraying, and FuseLink interruption-free relaying

ing buffer. *D2* introduces additional data copy across GPUs, limiting the throughput over indirect NICs.

② Low message latency over indirect NICs: *D1* achieves lower message latency than *D2* over indirect NICs without additional CPU involvement. Unlike *D2* CPU-initiated relay copy, where the GPU must synchronize with CPU to signal data readiness with extra delay, *D1* eliminates such synchronization overhead. Furthermore, *D1* masks the extra intra-server relaying latency by overlapping it with pipelined intra- and inter-server communication, effectively mitigating latency penalties for small-to-medium messages.

With benchmarking, we use remapped network buffer for efficiently relaying traffic to indirect NICs because of three key performance advantages: First, it has no duplicate data copy of the traffic, compared to CPU-initiated data copy between GPUs. Second, it has less device synchronizations because GPU threads can write data directly to peer GPUs. Third, it avoids CPU involvement, which saves the latency in calling memory copy functions.

4.2 Interruption-free Relaying

While intra-server relaying brings significant benefits in NIC throughput, it brings risk of interrupting peer GPUs. Specifically, multi-NIC communication may take peer GPU bandwidth resources, and the memory consumption on relay GPUs may drain peer GPU memories, causing unexpected out-of-memory (OOM) error.

Relaying without interrupting communication. FuseLink accelerates GPU communication only during idleness of the indirect NICs. Although equally spraying traffic to all NICs within the server achieves optimal NIC utilization, it may disturb or even interrupt peer GPU communication over direct NICs. FuseLink strikes a balance between NIC utilization and fairness. This ensures that peer GPU communication tasks are not interrupted by traffic relaying.

We compare FuseLink interruption-free design against traffic spraying in Figure 7, where two GPUs send data through two NICs respectively. The first GPU has a long inter-server flow, while the second GPU is the victim, which has a relatively short flow and is constantly interrupted by the first GPU. In contrast, FuseLink exploits the indirect NIC only when the second GPU does not have inter-server traffic, preventing unfairness problem. FuseLink controls the traffic volume during intra-server relaying to avoid interference with GPUs engaged in intra-server communication. In practice, intra-server GPU communication, such as aggregation in tensor parallelism, overlaps with inter-server communication via NICs. To ensure isolation, FuseLink marks NICs as busy during TP communication and relaying, preventing contention with intra-server communication and other relaying traffic.

Relaying without interrupting memory allocation. If memory requirements of the running tasks on relay GPUs are known, FuseLink can easily eliminate OOM risks by limiting the total sizes of relay memories. Unfortunately, accurately calculating the memory footprint of a running task on GPUs remains challenging, and existing works do not have precise estimations [19], leaving risks of causing OOM due to relay buffer allocations. To mitigate this problem, FuseLink provides a best-effort solution that combines both memory usage constraints and adaptive approaches.

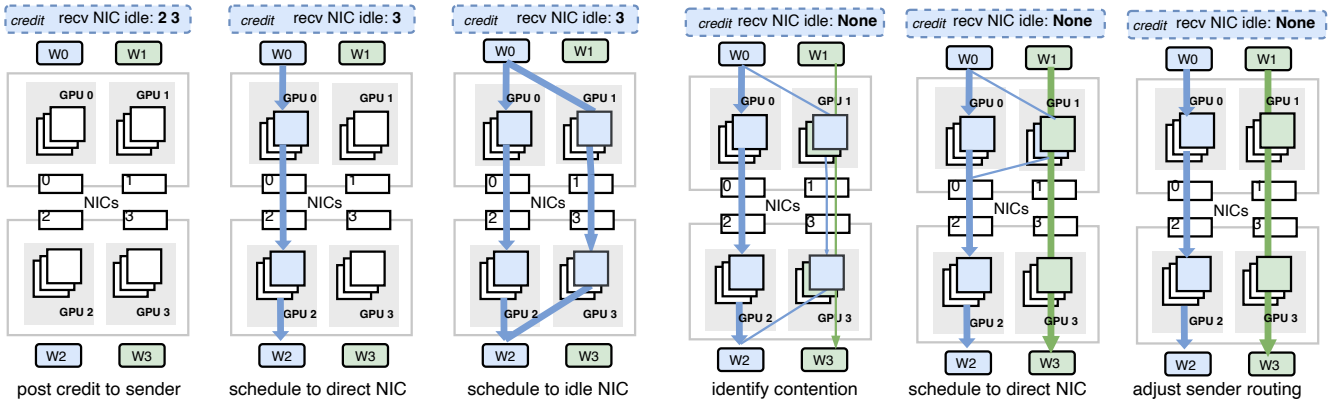
First, FuseLink sets a configurable upper limit for relay memory usage. If the relay memory on a GPU exceeds the limit, FuseLink will stop allocating more relay memories until memories from existing connections are released. Although this limitation cannot fully eliminate OOM risks, it allows FuseLink to adjust resource allocation according to user demands. For example, one may set high memory limitations to encourage FuseLink to take more multi-NIC transmissions with GPU relaying, thereby aggregating higher bandwidth with multiple NICs.

Second, FuseLink allows for prioritizing memory demands of running tasks on relay GPUs by removing relay memories when releasing can help satisfy memory allocation of the running tasks on relay GPUs. Specifically, when memory drains, FuseLink first checks whether releasing relay memories will free up enough space. If so, FuseLink releases the relay memory to give in to ongoing tasks. Otherwise, it means the task will encounter an OOM error regardless of the relay memory.

4.3 NIC Contention Mitigation

We next discuss the challenges on how to efficiently exploit idle NICs while avoiding GPU workers contending for NIC resources. Doing so is difficult mainly because of the unpredictable and imbalanced traffic in ML applications with dynamic communication patterns.

FuseLink mitigates NIC contention by assigning varying priorities to traffic from different GPUs during NIC scheduling. FuseLink monitors NIC load status and marks NIC as busy or idle, indicating whether NICs are working for high-priority GPUs. When a GPU engages in communication, it is granted the highest priority on its direct NIC, preventing peer



(a) Scheduling for exploiting idle NICs

(b) Scheduling for preventing contention

Figure 8: Traffic scheduling illustration. We emit ordinary elements in credits, such as memory regions, for simplicity

GPUs from using that NIC. Furthermore, FuseLink limits the outstanding traffic scheduled to idle NICs, thereby reducing potential contention when high-priority GPUs initiate communication. Our approach is based on the need for GPU workers to fully utilize their direct NICs in a 1:1 GPU-to-NIC configuration.

During communication, receivers encode idle NICs in credits and post to senders, who decide which NIC to transmit data based on NIC status of both sides. This design can be decomposed into two modules: 1) §4.3.1 FuseLink monitors NIC load status by inspecting completed network operations of high-priority GPUs. 2) §4.3.2 FuseLink aggregates idle NICs on both sides to guide traffic scheduling, with intra-server routing configured via memory remapping.

4.3.1 Worker-aware NIC Monitoring

FuseLink marks the status of NICs as idle or busy by monitoring the network operations posted by ML workers within the server. ML workers post network operations, such as send and receive, to FuseLink through first-in-first-out (FIFO) queues, instead of operating on RDMA connections directly. FuseLink collects network operations and post work requests to NICs. Then FuseLink polls RDMA completion queues to get finished operations.

The straightforward solution to decide whether a NIC is idle is by checking NIC TX and RX counters. However, reading counters cannot tell which worker is sending or receiving data, which hinders contention prevention when we have multiple workers transmitting data through the same NIC. Instead, FuseLink monitors NICs workload from the work requests on RDMA connections. Note that we configure NIC status to be decided by workers that have the highest priorities on the NIC. The rationale is that we aim to isolate traffic and guarantee the network performance of the high-priority worker.

FuseLink identifies idle NICs by periodically checking new completions of network operations posted by high-priority workers. If no new completion since last polling, FuseLink will mark the NIC as idle. As illustrated in Figure 8a, NIC1 is idle because W1 is not sending data to W3, and NIC0 is busy since W0 is sending to W2. Once found idle NICs, FuseLink schedules traffic from other workers to idle NICs. If a new completion is found, it means there exists new traffic since last time, and FuseLink will mark the NIC as busy. In this case, we will not schedule traffic to the busy NIC. FuseLink schedules traffic of low-priority workers to their direct NICs for performance isolation.

The rationale of marking NIC status as idle or busy is that ML applications typically have large traffic that can fully occupy NICs when they have communication tasks [34]. Therefore, we allow GPU workers to fully occupy their direct NICs. If GPU workers share direct NICs, FuseLink does not isolate traffic between these workers.

4.3.2 Load-aware Scheduling

We introduce load-aware traffic scheduling in three steps: 1) aggregating NIC load status on the sender and receiver side, 2) selecting NICs and initiating data transmission, and 3) performing intra-server traffic routing inside receivers via memory remapping.

Aggregating NIC load status. FuseLink orchestrates communication by receivers giving credits to senders and FuseLink configures credits to include NIC load status on receivers. Posting credit is necessary in RDMA communication because of the nature of RDMA that bypasses remote side processors, making senders unaware of receiver resources. RDMA communication with credits is common in existing distributed ML frameworks [1,45]. After getting credits, senders know NIC load status on both sides.

Selecting the NIC for sending. The NIC selection is based on NIC status and priorities of the worker on NICs. FuseLink selects the direct NIC of worker if it is idle. Otherwise, FuseLink selects an idle indirect NIC if exists. Note that we limit the number of outstanding operations posted to indirect NICs to reduce contention risks. Without limitation, we may post too

many requests to indirect NICs and cause severe contention when high-priority workers start communicating. If no idle NICs found, FuseLink selects the direct NIC.

Routing traffic on receivers. FuseLink writes data to receiver memory that resides on the GPU with direct PCIe connection to ensure transmission efficiency. However, the GPU could be different from the worker GPU that requests the receiving. In this case, FuseLink maps the receiving address to the memory where the received data locates. On sender side, the data may be already on relay buffers when getting credits. FuseLink remaps relay memory before the next sending operation, with the cost of one suboptimal sending.

We demonstrate the traffic scheduling with an example, as shown in Figure 8a. FuseLink first identifies NIC idleness when W1 is not posting any traffic. Then, for network operations posted by W0, FuseLink schedules them to NIC1 by allocating credits that indicate sending through NIC1. FuseLink remaps receiving memory on GPU1 to enable efficient NIC access. On sender side, FuseLink gets credits and posts sending operations to NIC1. Because the data is already on GPU0, the data is sent through suboptimal data path. After sending finished, FuseLink remaps relay buffers to GPU1 so that the subsequent sending is efficient.

When traffic contention happens on NIC1, as shown in Figure 8b, FuseLink detects contention when W1 is posting network operations to NIC1 and gets completions. In this case, FuseLink performs similar steps as exploiting idle NICs. FuseLink schedules subsequent network operations back to NIC0, together with router memory remapped to optimize data path.

4.4 Scheduling with Efficiency

FuseLink schedules communication to multiple NICs when detecting idle NICs in the server. Our design of NIC monitoring and traffic scheduling do not enforce strict idle NIC utilization or precise contention prevention, which is difficult to achieve due to control plane efficiency requirements. Instead, we have the following tradeoffs for control plane efficiency:

First, FuseLink marks NIC load status based on new completions of operations posted by GPU workers, which is later than transmission start time. The delayed marking is more efficient than detecting whether a NIC is sending traffic for a GPU worker. The delay is acceptable because existing frameworks, such as NCCL, divide large messages into chunks and send in pipeline. In NCCL, the default configuration is 512KB, which makes the delay marking about 10us under 400Gbps network.

Second, FuseLink allows bounded contention over limited number of network operations. In the worst case, FuseLink schedules a batch of operations to an idle indirect NIC and the peer worker starts communicating immediately. This ensures the scalability of FuseLink traffic scheduling in a large GPU cluster because FuseLink only needs to gather NIC utilization information of peer nodes during communication. The scheduling overhead is determined by the number of NICs and concurrent connections, which are limited by hardware configurations and ML applications.

Third, FuseLink allows bounded suboptimal transmission through indirect NICs, because the NICs selected may have suboptimal data path to the router GPUs, as illustrated in Figure 8b during receiver scheduling. We bound the suboptimal transmission by sender remapping, with the assumption that the next sending operations have optimal data path. Our design is based on the ML traffic pattern that NICs load status do not change frequently, as ML computation and communication last for long time intervals.

After optimizations, control plane overhead mainly comes from the following components:

NIC monitoring overhead. Before issuing operations, FuseLink checks NIC status and decides how to route the traffic. To reduce the monitoring overhead, FuseLink checks status of NICs after issuing a batch of network operations. For example, FuseLink checks NICs status after scheduling eight network operations. Subsequent operations are scheduled according to the NIC status after the last batch. The coarse grained NIC monitoring brings acceptable scheduling overhead because ML applications usually have large message sizes and are bounded by bandwidth.

Traffic routing overhead. When NIC encounters idleness or contention, FuseLink remaps the receiver relay buffer and switches the connection by posting credits to senders. The overhead consists of checking the NIC status, fetching new connection, and remapping relay buffers. Traffic rerouting happens when NIC status changes. In machine learning, NIC usage displays typical on-off patterns. Thus, NIC status will not change frequently, bringing limited traffic routing overhead.

5 Implementation

We implement FuseLink as an independent networking module to replace the default Infiniband networking in NCCL, a widely-used GPU communication library, so that ML applications can use FuseLink without modifying codes. We incorporate FuseLink in NCCL by intercepting networking layer functions called by NCCL proxy threads that originally post RDMA work requests to NICs. After loading the plugin, NCCL proxy threads interacts with FuseLink for GPU communication, instead of operating on NICs directly. Our implementation has about 3000 lines of code in C++.

In the default implementation, NCCL sets up worker threads as proxies to communicate with peer GPUs through inter-server network. Proxy threads build RDMA connections and post work requests to NICs. ML processes send data by writing to network buffers handled by proxy threads. When

receiving data, ML processes indicate receiving buffers and get notified from proxy threads when the data is ready.

We intercept proxy thread function calls to interact with FuseLink, including connection establishment, network buffer registration, and sending/receiving data.

Connection establishment. Instead of building raw RDMA connections, we modify proxy threads by posting connection request to FuseLink, which handles the request by setting up connections through multiple NICs within the server. After connection setup, proxy threads get connection ids that are used to post sending and receiving operations. All network operations are conducted by posting control messages to FuseLink with the connection id.

Registering buffers. By default, NCCL registers buffers on the specified device to enable RDMA NICs to access network buffers. We modify proxy threads to expose network buffers to FuseLink, which remaps buffers to relay GPUs and registers network buffers on RDMA NIC. Since each buffer only needs to be registered once in the NIC, remapping to relays brings minor overhead. Note that FuseLink maps network buffers back to original GPUs when NIC contention arises. Thus, we register network buffers on relay GPUs and original GPUs to avoid repeated registration. The registered memory regions are selected based on the current GPU to which the network buffer is mapped.

Sending & receiving. In NCCL, high-level communication tasks, such as sending a large data block and collective communication, are divided into data chunks.

On the sender side, the send function returns with an empty handler if no new credit is received. After receiving valid credits, NCCL sends with data in a ring slot and connection specified in credits. If the ring slot buffer is on a GPU that is not directly connected to the selected NIC, it means the receiver has scheduled an indirect NIC for data transmission. FuseLink marks the indirect NIC and remaps data to the buffer on a relay GPU. Send function returns with a valid handler if the data transfer to relay GPUs has started successfully. After the data is ready on the relay buffer, the sender starts data transfer to the remote side with RDMA.

On the receiver side, FuseLink collects NIC usage status to identify idle NICs and contention NICs. For each channel, FuseLink assigns one NIC to process receiving requests by giving credits to sender side specifying connections on the NIC. FuseLink schedules traffic to connections on other NICs by changing the RDMA Queue Pairs (QPs) and the memory addresses in ring slots used to receive data, which is transparent to applications.

6 Evaluation

We evaluate FuseLink capabilities in microbenchmarks and ML applications. In microbenchmark, we answer the following questions: 1) What is the maximum inter-server bandwidth for a single GPU with FuseLink compared to the default GPU-NIC binding? 2) What is the overhead of FuseLink in traffic scheduling? 3) What are the performance benefits of FuseLink on machine learning tasks with imbalanced traffic?

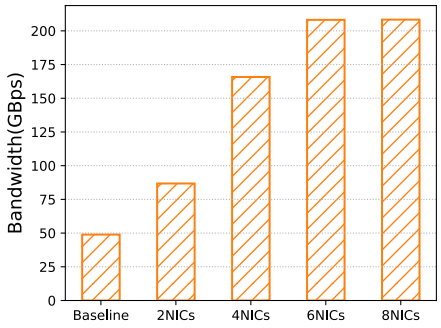


Figure 9: Average inter-server GPU bandwidth achieved by a single GPU when using different numbers of NICs

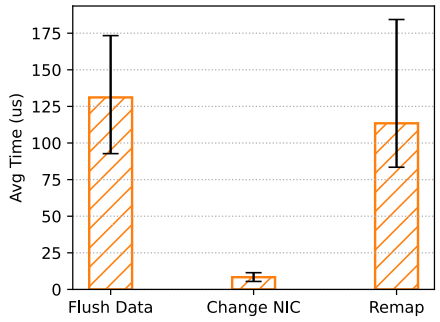


Figure 10: Average scheduling overhead decomposed into flushing data, changing NIC, and remapping

Evaluation setup. We conduct experiments on servers equipped with Intel 8480C CPU, eight Nvidia Hopper GPUs, and eight Connect-X7 400Gbps NICs for inter-server GPU communication. Intra-server GPUs are connected through eight-lane NVLinks and NVSwitches, delivering up to 200GB/s bandwidth for intra-server GPU communication. Each NIC is connected directly with one GPU through PCIe bridge, respectively. We deploy by running FuseLink scheduler as a daemon process and ML workers load FuseLink as a NCCL plugin to replace the default IB network.

Baselines. We use NCCL with PXN enabled as the baseline when performing inter-server GPU communication. NCCL with PXN automatically pick up the best path for inter-server GPU communication via both NVLink and PCIe, but is limited by statically bind traffic to paths and cannot support transmitting data through NVLink on receivers.

6.1 Microbenchmark

Bandwidth improvement. We evaluate how FuseLink can improve bandwidth when transmitting data between two inter-server GPUs. We divide the whole sending data into 512K messages, a common size for inter-server GPU communica-

Controls	Avg. cost(us)
post & pull ops	0.8-1.4
query NIC load	0.9-1.6
process send	2.8-3.5
process recv	4.9-5.6

Table 3: Network operation processing overheads in FuseLink

tion in NCCL, and post them to FuseLink controller. In the baseline experiment, the traffic is statically bound to one NIC for inter-server communication. By contrast, FuseLink sends data through direct NICs and indirect NICs with GPU relays. We change the maximum number of NICs used by FuseLink and record the average bandwidth.

As shown in Figure 9, FuseLink achieves up to 212GBps bandwidth with six NICs, compared to the default setting with about 50GBps throughput, where GPUs communicate through only one NIC. FuseLink gets higher bandwidth by utilizing more NICs during inter-server communication. The upper limitation depends on the bottleneck between GPUs and NICs. If the NVLink bandwidth is larger than the sum of bandwidths of indirect NICs, FuseLink can aggregate all NICs bandwidth at maximum. Otherwise, the maximum bandwidth is the sum of the direct NIC bandwidth and NVLink bandwidth. In this experiment, FuseLink bandwidth stops scaling because NVLink has reached the maximum throughput.

Scheduling overhead Compared to posting RDMA requests to NICs directly, FuseLink incurs scheduling overhead during monitoring NIC workload and determining the NIC to post RDMA operations (§4.3). To improve scheduling efficiency, we make tradeoff to allow bounded suboptimal scheduling and sender data path (§4.4). We show that the scheduling overhead is acceptable and the tradeoff will not harm the overall performance. Table 3 shows the overhead of FuseLink when processing RDMA operations and scheduling. NIC monitoring and NIC selection are implemented using shared memory, which brings 0.9-1.6us latency on a batch of network operations. Relay remapping is called when optimizing the data paths between NICs and GPUs, which brings around 95-193us latency in each remapping.

Figure 10 shows the scheduling overhead when FuseLink changes NIC to prevent contention. Before changing NICs, FuseLink flushes the network buffer, i.e., finish all network operations on the buffer, and finally performs buffer remapping to adjust intra-server traffic relaying. The total scheduling overhead is minor compared to the communication time in distributed machine learning.

6.2 End-to-End Evaluation

We demonstrate that FuseLink can effectively improve the NIC utilization in imbalanced traffic ML tasks and bring benefits to the overall performance.

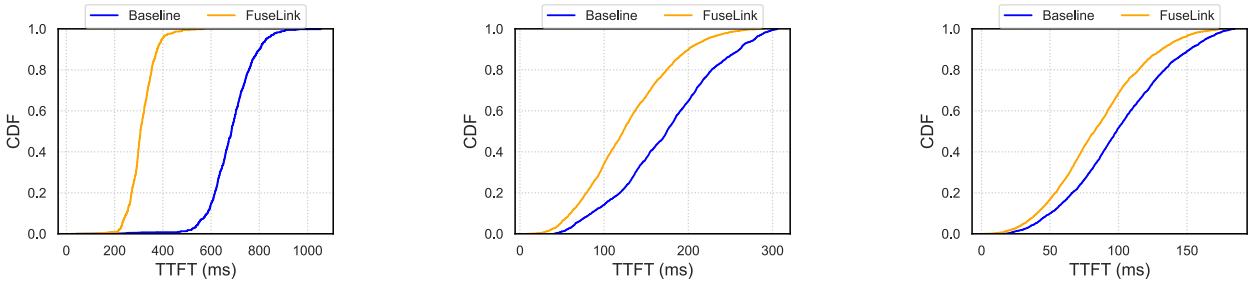
Model serving. We test FuseLink performance in model serving in disaggregated setting [47,69], where the serving process is divided into a prefill phase and a decoding phase across inter-server GPUs connected by NICs. We test OPT [67] models serving with 30B parameters under different tensor parallel degrees. We consider model serving in mixed requests, where batched user requests are fed to GPU groups for the prefill-stage computation. The processed user requests are sent to the decode-stage GPUs. The communication cost consists of sending prefix cache and user requests cache. We record the time-to-the-first-token (TTFT) as the performance metric, which includes requests computation and inter-server communication time.

#Instance	Setting	P50	P99
8	NIC Binding	684.54 ms	903.20 ms
	FuseLink	308.48 ms	464.55 ms
4	NIC Binding	174.46 ms	297.49 ms
	FuseLink	122.61 ms	259.73 ms
2	NIC Binding	98.09 ms	175.36 ms
	FuseLink	81.97 ms	160.36 ms

Table 4: Model serving TTFT comparison under different number of serving instances within a server

Figure 11a- 11c shows the CDF graph of TTFT recorded with different number of independent serving instances within a server. For every deployed serving instance, we randomly sample the serving prompt and feed to the prefill instances. We sample request lengths from public traces [5] with prefix and simulates with Poisson arrival rates. FuseLink shows 1.04-2.73× speedup over the baseline with NIC binding, where GPUs communicate only through the direct NICs. Notably, with smaller TP degrees and more instances per server, FuseLink shows greater speedup compared to scenarios with fewer instances and larger TP degrees, as the NIC exhibits more dynamic traffic pattern under more instances within a server. The performance improvement is attributed to two factors: First, FuseLink accelerates inter-stage data transmission, reducing communication overhead during first token generation; Second, FuseLink improves serving throughput and reduces the requests queueing time. Table 4 shows the TTFT of 50th percentile and 99th percentile of different numbers of serving instances within a server.

EP training. We test expert-parallel MoE training of Mixtral 8×22B model using tensor parallel degree as four and expert parallel degree as eight. In this case, each server has two experts training in parallel. We record the time spent on training iterations. Figure 12 shows the iteration times of the baseline and FuseLink. FuseLink improves the training throughput by 1.3× compared to the baseline. We notice that FuseLink brings less performance gain in later iterations, which can be explained by the design of gate layers that try to achieve load balance among experts [55], making the traffic across NICs closer to balanced NIC load compared to earlier iterations.



(a) Eight instances with TP=1 (b) Four instances with TP=2 (c) Two Instances with TP=4

Figure 11: FuseLink model serving performance under different TP degrees and number of serving instances

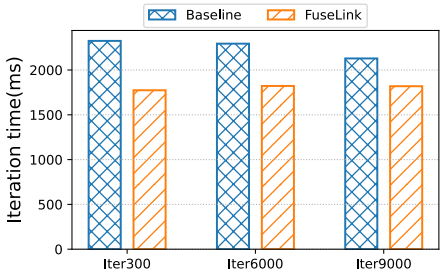


Figure 12: EP training times under NIC binding and FuseLink

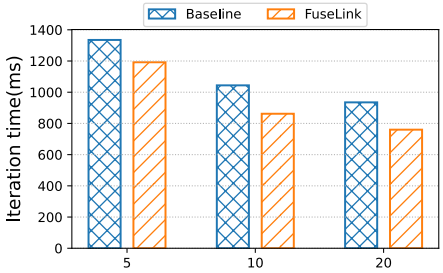


Figure 13: DLRM training iteration times under different cache sizes (GB)

DLRM training. We test FuseLink performance in DLRM training [13] model on Criteo advertisement dataset [4]. We configure 32 GPU workers to train the model in data parallelism and setup a dedicated server with large memories to maintain the embedding table. In each training iteration, the GPU workers first convert the categorical features in training samples into embedding vectors in the table. If the embedding cache does not have the required vector, the GPU workers need to fetch the embedding from the embedding server through NICs. Communication cost is mainly affected by the embedding cache sizes and training batch sizes of each worker. Limited cache sizes and large batch sizes tend to bring higher embedding cache miss and more embedding transmission. We set the training batch size as 1024 as suggested by [10]. Figure 13 shows the average iteration durations under different cache sizes when using FuseLink and static traffic binding to direct NICs. FuseLink can effectively reduce the training iteration times by accelerating embedding transmissions.

7 Discussion and Limitations

We discuss the application scope of FuseLink and limitations in existing ML infrastructures.

FuseLink in collective communication. FuseLink works the best under traffic imbalance scenarios, which are common in point-to-point communication. The point-to-point communication has become one of the major network costs in distributed machine learning, including MoE and model serving. In contrast, collective communication like ring all reduce synchronize GPUs with uniform traffic pattern across NICs [34], making them less suitable for FuseLink without extra modifications. Applying FuseLink in collective communication may require adjusting worker placements. For example, co-locating multiple data parallel groups within a server can produce imbalanced traffic across NICs in different parallel groups. However, such optimizations would require additional efforts to integrate with existing ML frameworks.

FuseLink in general GPU interconnections. FuseLink is designed and implemented on Nvidia GPU with NVLink connections, but is not limited by specific GPU devices, interconnections, or GPU-NIC ratios. In essence, FuseLink unlocks the capability of intra-server traffic routing and efficient multi-path transmission during inter-server GPU communication. FuseLink is powered by the ever-growing performance of dedicated GPU connections that have become integral in ML infrastructures. Industrial practices have deployed various of GPU interconnect [20, 46, 58] in their GPU clusters, making FuseLink widely applicable in today and future ML facilities [35, 36].

FuseLink ensures its design generality by relying on standard communication primitives. Specifically, RDMA enables NICs to access application memory from virtual address with backend physical memory mapped to any virtual memory on demand. We exploit this feature to enable RDMA NICs to access remapped location without changing the application network buffer address. We also take advantage of the GPU memory addressing that enables peer-to-peer communi-

cation between GPUs, which is essential in dedicated GPU connection to ensure efficient data transmission. The unified addressing and standard RDMA primitives do not rely on specific devices or GPU-NIC ratios. Therefore, FuseLink can be applied in general machine learning servers, including those with asymmetric GPU interconnects [8, 30] and other GPU-NIC ratios.

Corner cases of FuseLink scheduling. Since FuseLink optimizes scheduling efficiency with bounded contention, corner cases may arise where one GPU sends a small chunk of data through its direct NIC only when a peer GPU has just started using the NIC. In this scenario, FuseLink will route other GPU traffic to this idle NIC and then be preempted by the GPU with higher priority, and subsequently start using the idle NIC again in the next batch of sending. However, such cases are rare in machine learning workloads, because they feature large messages and relatively long time intervals between communication phases. This leaves sufficient time intervals for FuseLink to schedule traffic while maintaining relatively stable NIC utilization, minimizing routing frequency.

8 Related Work

PCIe $\times$ NVLink (PXN). NCCL has introduced PXN [44], an inter-server GPU communication mechanism that allows senders to use intermediate GPUs for inter-server communication. Similar to FuseLink, PXN removes the bottleneck of GPU-NIC connection that traverses through QPI/UPI, which cannot deliver full bandwidth. PXN is designed to avoid cross-rail traffic in rail-optimized GPU clusters because cross-rail traffic has to be routed through spine switches, which risks interfering with other flows in the network.

Despite the similarity of using intermediate GPUs, FuseLink has three key differences compared to PXN. First, PXN binds traffic to intermediate GPUs and statically copies data to GPUs. This can only improve one NIC throughput, yet others are in suboptimal PCIe connections. In contrast, FuseLink dynamically integrates NVLink interconnect in the network data path. Consequently, FuseLink can fully eliminate the suboptimal PCIe data path on multiple NICs during inter-server communication. Second, FuseLink identifies and leverages idle NICs, while PXN is configured statically by NCCL, given the system topology before running communication tasks. Finally, FuseLink achieves higher inter-server bandwidth for GPU communication by utilizing multiple NICs, while PXN is bounded by the single NIC bandwidth.

Multi-Path network protocols. Researchers have proposed datacenter networking protocols that aim to improve network resource utilization. For example, MP-TCP [63] and MP-RDMA [38] are multi-path transport protocols designed to utilize the rich path resources in datacenter networks; NetChannel [12], a host network stack that disaggregates the dedicated data path to improve utilization of host network resources. These protocols and network systems focus on a similar problem that network resources, such as host data paths and network paths, are not fully utilized because of dedicated resource assignments.

FuseLink differs from these works on the network resources and methods for multiplexing. On one hand, existing works try to multiplex network data paths with only host memories, while FuseLink mainly focuses on inter-server communication on GPU memories, which requires considering PCIe bandwidth and topology. On the other hand, FuseLink exploits dedicated intra-server GPU connections and integrates into inter-server networking to enhance inter-server networking, which is not explored in existing works.

Data path optimization. There have been efforts to improve network resource utilization from host side by solving the best data paths for inter-process communication. For example, SocksDirect [31] improves inter-process communication efficiency by implementing efficient communication primitives and automatically schedules the best data path to carry out data transmission, such as shared memory and RDMA. ARK [24] accelerates GPU communication by implementing an efficient DMA controller operated by GPUs, together with a GPU-driven execution model to improve computational efficiency without interfering with networking. FuseLink has similar methodologies that improve data path efficiency during GPU communication.

However, existing works on optimizing GPU communication data paths cannot be applied in dynamic ML tasks directly, because they cannot detect or leverage idle NICs timely. In contrast, FuseLink explored a new direction that exploit the underutilized NICs in dynamic-traffic ML tasks, with data path optimization in runtime.

9 Conclusion

In this paper, we present FuseLink to enable inter-server GPUs to efficiently communicate over multiple NICs. FuseLink exploits dedicated intra-server network for intra-server traffic routing to fully utilize GPU interconnects when communicating through NICs. FuseLink achieves improved inter-server bandwidth without interrupting ongoing computation and communication tasks on peer GPUs. Through our evaluation, we verify that FuseLink is effective in dynamic-traffic machine learning tasks, achieving high inter-server GPU bandwidth and accelerating machine learning tasks including LLM serving, EP MoE training, and DLRM training.

Acknowledgements

We thank our shepherd, Yang Zhou, and the anonymous OSDI reviewers for their insightful feedback. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITC ACCESS, NSFC 62402407, NSFC Excellent Young Scientists Fund Program (Overseas). Kai Chen is the corresponding author.

References

[1] Gloo: Collective communications library with various primitives for multi-machine training. URL: https://github.com/facebookincubator/gloo.

[2] NVIDIA DGX SuperPOD: Next Generation Scalable Infrastructure for AI Leadership. URL: https://docs.nvidia.com/https:/docs.nvidia.com/dgx-superpod-reference-architecture-dgx-h100.pdf.

[3] NVIDIA GB200 NVL72. URL: https://nvdam.widen.net/s/wwnsxrhm2w/blackwell-datasheet-3384703.

[4] Criteo . Criteo Display Advertisement Dataset. URL: https://go.criteo.net/criteo-research-kaggle-display-advertising-challenge-dataset.tar.gz.

[5] Microsoft . AzurePublicDataset. URL: https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md.

[6] Nvidia . Unified Addressing. URL: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__UNIFIED.html.

[7] PCI-SIG . PCI Express Base Specification. URL: https://pcisig.com/specifications/pciexpress/.

[8] WikiChip . Infinity Fabric (IF) - AMD. URL: https://en.wikichip.org/wiki/amd/infinity_fabric.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, USA, 2016. USENIX Association.

[10] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, February 2021. URL: http://dx.doi.org/10.1109/HPCA51647.2021.00072, doi:10.1109/hpca51647.2021.00072.

[11] Infiniband Association. Infiniband Roadmap. URL: https://www.infinibandta.org/infiniband-roadmap/.

[12] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards *M*s Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 767–779, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544230.

[13] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, Dlrs 2016, pages 7–10, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2988450.2988454.

[14] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K. (D K) Panda. NV-group: Link-efficient reduction for distributed deep learning on modern dense GPU systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, Ics '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3392717.3392771.

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

jin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. DeepSeek-V3 technical report, 2025. URL: https://arxiv.org/abs/2412.19437, arXiv:2412.19437.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2019. arXiv:1810.04805.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojeviÄĞ.

[18] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(1), January 2022.

[19] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating GPU memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Esec/Fse 2020, pages 1342–1352, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368089.3417050.

[20] Google. System architecture | Cloud TPU. URL: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm.

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215, 2016.

[22] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: A factorization-machine based neural network for CTR prediction, 2017. URL: https://arxiv.org/abs/1703.04247, arXiv:1703.04247.

[23] Mert Hidayetoglu, Simon Garcia De Gonzalo, Elliott Slaughter, Yu Li, Christopher Zimmer, Tekin Bicer, Bin Ren, William Gropp, Wen-Mei Hwu, and Alex Aiken. CommBench: Micro-benchmarking hierarchical networks with multi-GPU, multi-NIC nodes. In *Proceedings of the 38th ACM International Conference on Supercomputing*, Ics '24, pages 426–436, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3650200.3656591.

[24] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. ARK: GPU-driven Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 87–101, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/hwang.

[25] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. XDL: An industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, Kdd '19. ACM, August 2019. URL: http://dx.doi.org/10.1145/3326937.3341255, doi:10.1145/3326937.3341255.

[26] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/jiang.

[27] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou,

Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs, 2024. arXiv:2402.15627.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[29] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, Sosp '23, pages 611–626, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3600006.3613165.

[30] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern GPU interconnect: PCIe, nvlink, NV-SLI, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, January 2020. doi:10.1109/TPDS.2019.2928289.

[31] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter Sockets Can Be Fast and Compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 90–103, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342071.

[32] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, USA, 2014. USENIX Association.

[33] Weiqing Li, Guochao Jiang, Xiangyong Ding, Zhangcheng Tao, Chuzhan Hao, Chenfeng Xu, Yuewei Zhang, and Hao Wang. FlowKV: A disaggregated inference framework with low-latency KV cache transfer and load-aware scheduling, 2025. URL: https://arxiv.org/abs/2504.03775, arXiv:2504.03775.

[34] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. Understanding communication characteristics of distributed training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, pages 1–8, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3663408.3663409.

[35] Heng Liao, Bingyang Liu, Xianping Chen, Zhigang Guo, Chuanning Cheng, Jianbing Wang, Xiangyu Chen, Peng Dong, Rui Meng, Wenjie Liu, Zhe Zhou, Ziyang Zhang, Yuhang Gai, Cunle Qian, Yi Xiong, Zhongwu Cheng, Jing Xia, Yuli Ma, Xi Chen, Wenhua Du, Shizhong Xiao, Chungang Li, Yong Qin, Liudong Xiong, Zhou Yu, Lv Chen, Lei Chen, Buyun Wang, Pei Wu, Junen Gao, Xiaochu Li, Jian He, Shizhuan Yan, and Bill McColl. Ub-mesh: a hierarchically localized nd-fullmesh datacenter network architecture, 2025. URL: https://arxiv.org/abs/2503.20377, arXiv:2503.20377.

[36] Xudong Liao, Yijun Sun, Han Tian, Xinchen Wan, Yilun Jin, Zilong Wang, Zhenghang Ren, Xinyang Huang, Wenxue Li, Kin Fai Tse, Zhizhen Zhong, Guyue Liu, Ying Zhang, Xiaofeng Ye, Yiming Zhang, and Kai Chen. mFabric: An efficient and scalable fabric for mixture-of-experts training, 2025. URL: https://arxiv.org/abs/2501.03905, arXiv:2501.03905.

[37] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, Acm Sigcomm '24, pages 38–56, Sydney, NSW, Australia and New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3651890.3672274.

[38] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 357–371, Renton, WA, April 2018. USENIX Association. URL: https://www.usenix.org/conference/nsdi18/presentation/lu.

[39] Samuel Matzek, Max Grossman, Minsik Cho, Anar Yusifov, Bryant Nelson, and Amit Juneja. Data-parallel distributed training of very large models beyond GPU capacity, 2018. arXiv:1811.12174.

[40] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Distributed serving of large language models via max-flow on heterogeneous gpus, 2024. URL: https://arxiv.org/abs/2406.01566, arXiv:2406.01566.

[41] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359646.

[42] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Sc '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458817.3476209.

[43] NVIDIA. Common Data Center Architectures | Two-Tier Clos Architecture. URL: https://docs.nvidia.com/networking-ethernet-software/knowledge-base/Setup-and-Getting-Started/layer-1-Data-Center-Cheat-Sheet/#two-tier-clos-architecture-leaf-spine.

[44] Nvidia. Doubling all2all performance with NVIDIA collective communication library 2.12. URL: https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/.

[45] NVIDIA. Nvidia NCCL. URL: https://developer.nvidia.com/nccl.

[46] NVIDIA. NVLink & NVSwitch: Fastest HPC Data Center Platform. URL: https://www.nvidia.com/en-us/data-center/nvlink/.

[47] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *ISCA*, June 2024. URL: https://www.microsoft.com/en-us/research/publication/splitwise-efficient-generative-llm-inference-using-phase-splitting/.

[48] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Sosp '19, pages 16–29, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359642.

[49] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. Efficient inter-node MPI communication using gpudirect RDMA for InfiniBand clusters with NVIDIA gpus. In *2013 42nd International Conference on Parallel Processing*, pages 80–89, 2013. doi:10.1109/ICPP.2013.17.

[50] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for LLM serving, 2024. URL: https://arxiv.org/abs/2407.00079, arXiv:2407.00079.

[51] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, Santa Clara, CA, April 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/rajasekaran.

[52] Kiran Ranganath, AmirAli Abdolrashidi, Shuaiwen Leon Song, and Daniel Wong. Speeding up collective communications through inter-GPU re-routing. *IEEE Computer Architecture Letters*, 18(2):128–131, 2019. doi:10.1109/LCA.2019.2933842.

[53] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[54] Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018. arXiv:1802.05799.

[55] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL: https://arxiv.org/abs/1701.06538, arXiv:1701.06538.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2020. arXiv:1909.08053.

[57] Yiltan Hassan Temuçin, AmirHossein Sojoodi, Pedram Alizadeh, and Ahmad Afsahi. Efficient multi-path nvlink/pcie-aware UCX based collective communication for deep learning. In *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 25–34, 2021. doi:10.1109/HOTI52880.2021.00018.

[58] Ajay Tirumala and Raymond Wong. NVIDIA blackwell platform: Advancing generative AI and accelerated computing. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–33, Los Alamitos, CA, USA, August 2024. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/HCS61935.2024.10665247, doi:10.1109/HCS61935.2024.10665247.

[59] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models, 2023. arXiv:2302.13971.

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

[61] Steve Wang and Will Cukierski. Click-through rate prediction, 2014. URL: https://kaggle.com/competitions/avazu-ctr-prediction.

[62] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/wei.

[63] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association. URL: https://www.usenix.org/conference/nsdi11/design-implementation-and-evaluation-congestion-control-multipath-tcp.

[64] Kaiqiang Xu, Decang Sun, Han Tian, Junxue Zhang, and Kai Chen. GREEN: Carbon-efficient resource scheduling for machine learning clusters. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 999–1014, Philadelphia, PA, April 2025. USENIX Association. URL: https://www.usenix.org/conference/nsdi25/presentation/xu-kaiqiang.

[65] Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, Xinchen Wan, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. Design and operation of shared machine learning clusters on campus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, Asplos '25, pages 295–310, Rotterdam, Netherlands and New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3669940.3707266.

[66] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[67] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models, 2022. URL: https://arxiv.org/abs/2205.01068, arXiv:2205.01068.

[68] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 559–578. USENIX Association, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

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



Tigon: A Distributed Database for a CXL Pod

Yibo Huang, Haowei Chen, Newton Ni, Yan Sun†, Vijay Chidambaram, Dixin Tang, Emmett Witchel
The University of Texas at Austin †*University of Illinois Urbana-Champaign*

Abstract

Building efficient distributed transactional databases remains a challenging problem despite decades of research. Existing distributed databases synchronize cross-host concurrent data accesses over a network, which requires numerous message exchanges and introduces performance overhead.

We describe Tigon, the first distributed in-memory database that synchronizes concurrent cross-host data accesses using atomic operations on CXL memory. Using CXL memory is more efficient than network-based approaches, but Tigon's design must address CXL's higher latency and lower bandwidth relative to local DRAM, as well as CXL's limited hardware support for cross-host cache coherence. For TPC-C and a variant of YCSB, Tigon achieves up to 2.5× higher throughput compared with two optimized shared-nothing databases that use CXL memory as a transport and up to 18.5× higher throughput compared with an RDMA-based distributed database.

1 Introduction

Despite decades of research, efficiently scaling a transactional database to multiple hosts remains a challenging task due to the complexities of data synchronization and concurrency control across hosts [17, 33, 48, 49, 56, 61, 66, 77, 85]. Existing distributed databases synchronize cross-host concurrent data accesses over a network, which introduces high and variable overhead due to cross-host message exchanges and high network latencies. For example, traditional shared-nothing databases partition data across hosts and let each host process all read/write operations to its partition [17, 33, 56, 66]. The performance of this architecture drops significantly when the database needs to process a large number of *multi-partition transactions*, that is, transactions that need to read or write data in multiple partitions and will be executed by multiple hosts. Multi-partition transactions are costly due to numerous message exchanges between hosts and the two-phase commit (2PC) protocol necessary to complete them [23, 48, 49, 61, 77, 85].

A recent line of research leverages RDMA to accelerate distributed transaction processing [4, 13, 20, 21, 32, 62, 71–73]. Many papers adopt an RDMA-based shared and disaggregated memory architecture to avoid multi-partition transactions [19, 44, 75, 78–80, 87]. However, the latency for memory access through RDMA networks is one to two orders of magnitude higher than local DRAM [26] and synchronizing concurrent read/write operations to database tuples through RDMA networks is still expensive, because round trips on RDMA-based networks have latencies in the microseconds range.

This paper proposes a new direction for building distributed transactional databases: *instead of synchronizing cross-host concurrent data accesses over a network, we propose synchronizing these accesses directly through memory, leveraging the capabilities of emerging CXL technology*. CXL memory is a memory module that can be physically connected and shared by a small number of hosts (e.g., 8–16 [6, 45]). The CXL protocol allows CPUs to directly access CXL memory using normal load and store instructions, and it provides much lower latency compared with RDMA-based shared memory. With CXL version 3.0 (and in 3.2 [2], which is the current version as of this writing) the specification allows CXL memory to be shared across hosts with hardware cache coherence, which provides new opportunities for cross-host data sharing and data synchronization. We call a collection of machines that share CXL memory a *CXL pod* [11, 29, 74, 86].

While CXL memory has advantages over RDMA and other networking technologies, it must be used with care. CXL memory has two major limitations: 1) it has higher latency (214-394 ns vs. 111-117 ns) and lower bandwidth (18-52 GB/s vs. 218-246 GB/s for a read-only workload) compared with local DRAM, as measured on hardware prototypes in a recent study [47]. and 2) only a small part of the physical CXL memory space, ranging from dozens to hundreds of MBs, will be kept hardware cache-coherent as reported in recent work from AMD [30]. Higher latency and lower bandwidth means a database cannot simply place all its data in CXL memory if it wants to achieve peak performance. Limited hardware cache-coherent memory means that database synchronization structures must be reorganized to minimize the use of this memory.

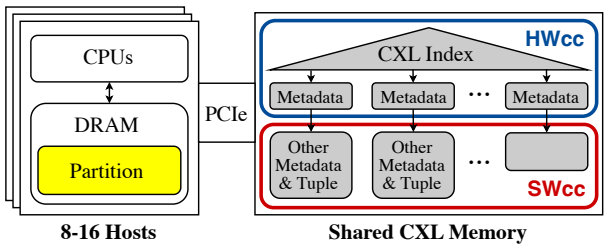


Figure 1: Tigon on a CXL pod.

In this paper, we introduce Tigon[1], the first distributed in-memory database that leverages CXL memory for efficiently synchronizing cross-host concurrent data accesses. Tigon benefits from the performance of CXL memory while avoiding CXL's weaknesses, based on a key insight about general transactional workloads: *while the size of a database could be large, the size of the set of tuples that will be concurrently read or written by running transactions from different hosts is small*. This is because each transaction typically accesses a small number of tuples (e.g., from a few to dozens) and the number of concurrently running transactions is typically equal to the number of available CPU cores for an in-memory database. For example, in TPC-C [7], a transaction accesses an average of 39 tuples, with a total data size of approximately 7 KB. Assuming 1,000 cores, each core executing one transaction, the system has at most 39K active tuples, which amount to 7 MB of data. We call such a set of tuples the *Cross-host Active Tuples* or CAT for short. We design Tigon to efficiently maintain the CAT in CXL memory, thereby increasing performance by converting multiple message exchanges into data structure operations. Because the data structure needs to contain only data that is currently shared, Tigon's performance is not limited by CXL's inferior latency and bandwidth relative to local DRAM.

Tigon initially partitions data across hosts as shown in Figure 1 and dynamically maintains the CAT in CXL memory. Cross-host concurrent accesses to the CAT can be efficiently synchronized using atomic operations, metadata (e.g., database latches), and hardware cache coherence, allowing Tigon to adopt a single-host concurrency control protocol to ensure transaction semantics without using 2PC. If one host needs to access a tuple stored in a remote host, Tigon moves the tuple to shared CXL memory. The data shared in CXL memory will be moved back to its original host based on a system policy that considers access patterns to the shared data. In addition, each host can retain the data accessed exclusively by that host in local DRAM, leveraging its low latency and high bandwidth.

Tigon addresses two broad challenges for maintaining the CAT in CXL memory. First, maintaining the CAT in the limited hardware cache-coherent region may introduce frequent data movement between local DRAM and CXL memory, increasing CXL memory bandwidth usage and the latency for processing a transaction. Tigon addresses this challenge with an efficient software cache coherence protocol that enables it to use CXL memory that is not kept coherent by hardware. Tigon uses the capacity of this large software cache-coherent region for database data, which reduces the need for data movement. Tigon also reduces data movement with common sense optimizations like only copying updated tuples from CXL memory back into local DRAM.

Indexes and certain metadata (e.g., database latches) require intensive synchronization across hosts, which is efficiently provided by atomic instructions operating on hardware cache-coherent memory. A key design principle for Tigon is to separate data by synchronization requirements: synchronization-heavy metadata (e.g., indexes and latches) is stored in the hardware cache-coherent (HWcc) region, while other data (such as tuples) is stored in the software cache-coherent (SWcc) region, as illustrated in Figure 1. Databases already have synchronization primitives to ensure the integrity of tuples under concurrent accesses, so we co-design Tigon's software cache coherence protocol with the database's mechanism for protecting the integrity of tuples to minimize synchronization overhead.

The second challenge Tigon addresses is to efficiently access data and provide transaction semantics (i.e., ACID properties) while data is moving between local DRAM and CXL memory. To do so, Tigon utilizes database latches and indexes to provide efficient access to the CAT under data movement and enhances concurrency control (i.e., two-phase locking (2PL) and next-key locking) and logging protocols to ensure transaction semantics without using 2PC. First, for a tuple moved to CXL memory, the owner host can cache a shortcut pointer to quickly find this tuple without searching the CXL index in CXL memory, reducing latency and CXL memory bandwidth usage. The challenge is to ensure its correctness under concurrent data movement. Second, we enhance 2PL to work together with a scalable logging protocol [82] such that Tigon, running on multiple hosts, does not need to use 2PC when committing transactions. Finally, we design an enhanced next-key locking protocol that maintains minimal additional metadata for each tuple to avoid the phantom problem [24]. We also design a lightweight data movement policy that keeps tuples that will most likely be exclusively accessed by a single host in local DRAM.

The novelty of Tigon lies in its use of inter-host CXL memory to accelerate transaction processing, while avoiding the performance pitfalls of using CXL memory naïvely. Tigon uses inter-host CXL memory to accelerate message passing, but it also reduces the need to pass messages by using data structures in inter-host CXL memory that are synchronized via atomic operations. The result is greater throughput for transaction processing, especially transactions that access data from multiple partitions. Experiments on TPC-C and a variant of YCSB demonstrate that Tigon achieves up to 2.5× higher throughput compared with two optimized shared-nothing databases using CXL memory as a transport and up to 18.5× higher throughput compared with an RDMA-based distributed database.

While prior research has explored improving the efficiency

[1] A Tigon is a hybrid of a male tiger and a female lion.

of databases using CXL memory [8, 10, 39, 60], none considers leveraging limited hardware cache-coherent memory for building an efficient distributed database. This work is also different from prior research on memory tiering and pooling [22, 45, 54, 59, 65], which makes CXL memory available to applications transparently, placing the burden of memory management and data migration to system software (or software and hardware [83]). Tigon, in contrast, is built to explicitly move data to and from CXL memory shared among multiple hosts and explicitly allocates CXL memory and places data structures in either the hardware or software cache-coherent regions.

This paper makes the following contributions:

- We introduce the first distributed in-memory database that synchronizes cross-host concurrent data accesses via atomic operations on CXL memory.
- We address the hardware limitations of CXL memory by maintaining the CAT in CXL memory and using a software cache coherence protocol to reduce the cost of maintaining the CAT (§3.2).
- We design new methods for efficiently accessing the CAT under data movement and enhance concurrency control and logging protocols to ensure transaction semantics without using 2PC (§3.4 and §3.5).
- We implement Tigon (https://github.com/ut-athenarc/tigon) and perform extensive experiments that show its performance advantages over shared-nothing databases and an RDMA-based distributed database (§4).

2 CXL Pod Background and Challenges

Compute Express Link (CXL) is a high-performance, open-standard interconnect for communications between CPUs, devices, and memory based on PCIe 5.0 and 6.0. CXL memory is a memory module connected to one or multiple hosts and has different characteristics depending on the CXL specification [2] version and the type of device. Type-3 CXL devices are memory expanders (that use the CXL.mem protocol), with the version 1.1 CXL specification allowing a single host to access PCIe-connected memory in a cache-coherent manner. CXL 3.0 and the most recent CXL 3.2 specify cacheline-granularity memory sharing and cache coherence across multiple hosts.

A recent study [47] shows that CXL memory has significantly higher latency (214-394 ns vs. 111-117 ns) and lower bandwidth (18-52 GB/s vs. 218-246 GB/s for a read-only workload) compared with local DRAM. Exact bandwidth depends on the specific hardware prototype and the memory access pattern. Another study [64] reports 11.7 GB/s sequential read bandwidth, but falls as low as ∼5 GB/s for other access patterns.

CXL pod. A CXL pod [11, 29, 74, 86] includes a small number of machines (e.g., 16) that connect directly to a shared CXL memory module via multiple ports (called a multi-headed device [2] (MHD)). The advantage of an MHD is that it allows multiple hosts to connect to a single memory device without a switch, which adds significant latency. This scale of configuration provides rich computing resources for transactional databases and has small latency and bandwidth penalties compared with a single host [45].

A CXL pod is an intermediate organization between a shared-memory multi-processor and a distributed system. The pod has hosts with local, hardware cache-coherent memory. Each host also connects to a single, shared CXL memory module. The inter-host CXL memory supports limited (and expensive) hardware cache coherence. There is a long debate in computer architecture as to the limits of scalability of SMP machines [36, 53], but the pod provides a new set of tradeoffs. We built Tigon to navigate this novel and complex tradeoff space.

CXL memory devices that can be used in a CXL pod exist, notably SK Hynix's Niagara 2.0 [6] and a Microsoft prototype [11]. The former is limited to 8 hosts, and the latter to 2. Neither supports hardware cache coherence, but their latency and bandwidth are broadly similar to our evaluation testbed (§4). Tigon is designed for CXL devices with some hardware cache-coherent memory. Designing a database to be efficient without hardware cache-coherent memory is worthwhile future work.

Comparison to NUMA and RDMA-based architectures. NUMA and CXL memory share some architectural properties, but detailed studies show that NUMA behavior often differs significantly from CXL [31, 64]. In our experience building Tigon, we found that a NUMA platform served as an informative proxy for CXL, but much of our system tuning was specialized to CXL (e.g., CXL bandwidth is much lower than NUMA bandwidth on our testbeds). RDMA-based distributed databases are more scalable in terms of the number of nodes in the system because they use explicit addressing and switches, but they have a higher latency, in the microseconds range.

Hardware limitations of CXL memory. Three properties of CXL memory make it challenging to use for a database: (1) it has higher latency than local DRAM; (2) it has lower bandwidth than local DRAM; (3) it has limited hardware cache-coherent memory capacity. The high latency and low bandwidth means any system that uses CXL memory must lean heavily on the performance of local DRAM. Data can be copied into and out of CXL memory, but too much data movement will overwhelm its limited bandwidth.

Why inter-host hardware cache coherence is limited. The CXL standard, starting from 3.0, provides support for inter-host hardware cache coherence through back-invalidations, that is, the CXL device includes a snoop filter (similar to processor snoop filters) for managing coherence. However, providing cache coherence for the entire CXL memory address space will be too expensive to be practical due to the size of the snoop filter required and the metadata overhead, as shown by AMD [30]. There is no theoretical reason for the limitation, it is purely practical. There is a fixed area budget for the CXL

controller that is consumed by ports and control logic [11]. Snoop filters for the entire CXL memory address space would be large because they must track all cacheable data for every host. As an extreme example, Intel's Granite Rapids 6980P processor would require tags for 16×504 MB, or 7.9 GB of cacheable data. Snoop filters that are cost effective will cover a much smaller region. In order to control costs and complexity, vendors will limit the size of the hardware cache-coherent region of CXL memory devices. Because we do not have exact figures, our study varies the amount of available hardware cache-coherent memory to evaluate its impact (§4).

Failure model. We assume a fail-stop model, and achieve durability using logging. Logs are written to local SSD devices. A failure of any component causes a failure of (and subsequent recovery for) the entire system.

3 Tigon Design and Implementation

Tigon is a distributed in-memory database designed to run transactions efficiently on a CXL pod. Unlike traditional distributed databases, Tigon exploits unique features of the CXL pod to lower the overhead of cross-host coordination and significantly increase transaction throughput.

3.1 Tigon Overview

Architecture. Tigon adopts the Pasha architecture [29], where data is initially partitioned across the hosts of the CXL pod and dynamically moved between local DRAM and CXL memory. Each host is the *owner* of a disjoint partition of the data. It stores its partition in local DRAM and accesses this partition with low latency and high bandwidth. When data is accessed by its owner host, no cross-host coordination is required. When a host wants to access data it does not own, it requests the owner to move the data to shared CXL memory such that Tigon can maintain the *cross-host active tuples* (CAT), the set of tuples that are concurrently read or written by active transactions running on different hosts, in CXL memory. Shared data requires synchronization across hosts; hosts synchronize using latches and locks in the hardware cache-coherent part of CXL memory. Hosts communicate with each other by sending messages using CXL memory as a transport, similar to HydraRPC [50].

Software cache coherence (§3.2)**.** Tigon introduces a novel software cache coherence protocol for CXL memory because most of the physical CXL memory space will lack hardware cache coherence support. The chief insight is that hardware support for cache coherence is *only required* for key metadata, such as latches and locks, that are frequently accessed by multiple hosts; by carefully limiting the size of such metadata and strategically placing it in the hardware cache-coherent region, Tigon can provide software cache coherence using the database's existing concurrency control techniques. Software cache coherence allows Tigon to use much more CXL memory than the limited hardware cache-coherent region.

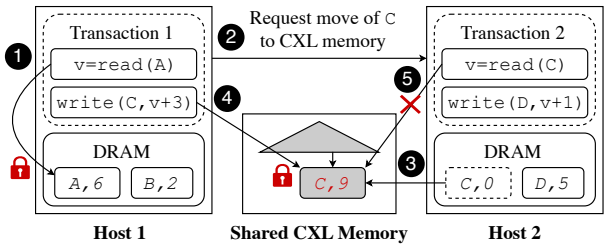


Figure 2: An example transaction workflow in Tigon.

Handling concurrency (§3.4)**.** Tigon needs to manage concurrency: both among threads within a host, and between hosts. Tigon utilizes database latches and indexes to efficiently access the CAT even when data is dynamically moving between local DRAM and CXL memory. Tigon adapts concurrency control protocols, two-phase locking (2PL) and next-key locking, to ensure serializability while maintaining the CAT.

Avoiding two-phase commit (2PC) (§3.3 and §3.5)**.** 2PC adds considerable overhead to transactions as it requires two rounds of message exchanges. Avoiding 2PC requires a single host to execute and log all database modifications involved in a transaction, enabling a host to commit its transaction locally. Tigon avoids 2PC based on two insights. First, by maintaining the CAT in CXL memory, a single host can complete all modifications to database tuples required by a transaction (e.g., by taking locks on CXL-resident structures). Second, although transaction execution in Tigon may involve modifying indexes of other hosts, indexes can be *reconstructed* from the database tuples during recovery, avoiding the need to log index modifications. As a result, only tuple changes need to be logged at the host executing the transaction to ensure atomicity and durability. Combining these two insights, a single host executes all transaction operations (excluding data movement and modifications to indexes of other hosts) and logs tuple modifications. We adapt a logging protocol [82], which was designed for scalability, to avoid 2PC.

Example transaction workflow. Figure 2 shows an example transaction workflow in Tigon. The database stores key-value pairs (e.g., tuple (A, 6) has key A and value 6). The example transaction (transaction 1) reads tuple *A* and writes tuple *C*, and is executed by a worker thread on Host 1 (its transaction worker).

1. The transaction worker acquires the read lock for tuple (A, 6) using 2PL. It then reads tuple (A, 6).
2. To write to tuple (C, 0), which is stored in Host 2, the transaction worker sends a message to Host 2 to request a move of (C, 0) to CXL memory.
3. A worker thread on Host 2 moves (C, 0) to CXL memory.
4. The transaction worker acquires the write lock on tuple (C, 0) and updates it to (C, 9).

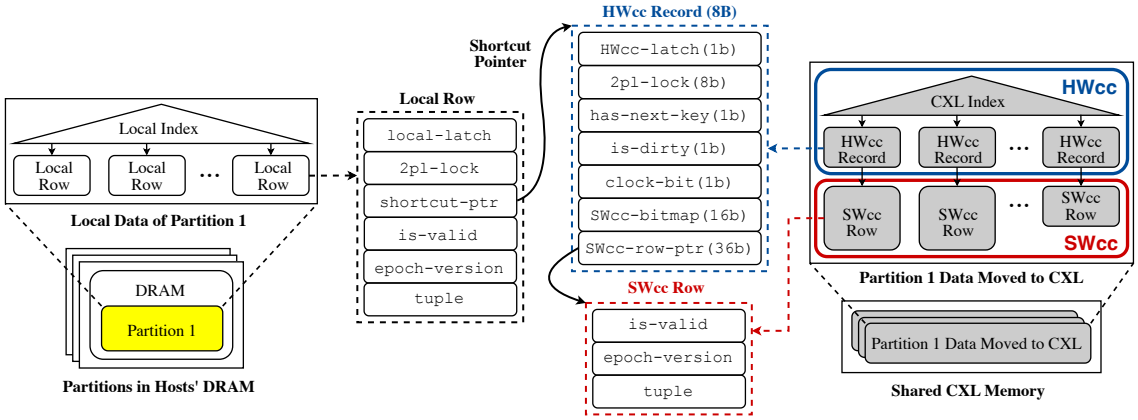


Figure 3: Data organization in Tigon.

5. The transaction worker of transaction 2 needs to access the tuple for key C at the same time; it finds (C, 9) in CXL memory is locked and aborts transaction 2 to prevent deadlocks (using the NO_WAIT policy).

After the transaction worker finishes transaction 1, it commits the transaction without any further messages to Host 2 or using 2PC, since it executes all operations of transaction 1 itself and has all the log records of the changes transaction 1 has made to the database. In this example, the CAT is the single tuple with key C.

3.2 Data Organization & SW Cache Coherence

We now describe how Tigon organizes data and implements software cache coherence. Figure 3 shows an overview of the data organization in Tigon.

Partitions and local row. Tigon organizes data into tables and partitions them across hosts. Tigon stores the partitioning function in a shard map and replicates it across hosts (not shown in the figure). Each host uses the shard map to find the host that owns a certain tuple, as do many partition-based distributed databases [1, 18, 33, 77]. Partitions are initially stored in each host's local DRAM. Data is stored in the form of *local row*. Each local row contains the following:

- Latch for mutual exclusion (`local-latch`)
- Locking metadata for 2PL (`2pl-lock`)
- Pointer to the tuple in CXL memory (`shortcut-ptr`)
- Flag indicating whether the tuple is valid (`is-valid`)
- Field used for logging (`epoch-version`)
- Database tuple (`tuple`)

Data and metadata in CXL memory. Tigon maintains the CAT in CXL memory. To minimize hardware cache-coherent (HWcc) memory usage while still enabling efficient cross-host synchronization on tuples moved to CXL memory, Tigon stores the metadata needed for intensive cross-host synchronization in the HWcc region and everything else (i.e., databse tuples and other metadata) in the CXL memory region that is not hardware cache-coherent (non-HWcc). If a local row is moved to CXL memory, Tigon divides it into an *HWcc record* stored in the HWcc region and an *SWcc row* stored in the non-HWcc region. We develop a software cache coherence protocol to enable cacheable reads of the SWcc rows.

HWcc record and SWcc row. Each SWcc row contains the `is-valid`, `epoch-version`, and `tuple` fields, similar to a local row. Each HWcc record occupies 8 bytes and contains the following fields:

- 1-bit exclusive latch for mutual exclusion (`HWcc-latch`)
- 8-bit locking metadata for 2PL (`2pl-lock`)
- 1-bit flag supporting next-key locking (`has-next-key`)
- 1-bit flag indicating whether the tuple has been modified since being moved to CXL memory (`is-dirty`)
- 1-bit supporting the CLOCK policy (`clock-bit`)
- 16-bit bitmap for software cache coherence (`SWcc-bitmap`)
- 36-bit pointer to SWcc row (`SWcc-row-ptr`)

The `SWcc-row-ptr` stores an offset relative to a base memory pointer. It can represent up to 64 GB of memory space, and is large enough for storing the data that needs to be frequently accessed across hosts in a transactional database. If the offset is defined at the granularity of cachelines, the addressable tuple storage expands to 2 TB (i.e., $2^{36} \times 2^{6}$ bytes).

The `is-dirty` bit allows the owner host to read clean tuples from local DRAM instead of CXL memory, significantly reducing CXL memory bandwidth usage.

Indexes and shortcut pointer. Tigon maintains two types of indexes: local index and CXL index. Each host maintains a local index in DRAM, which maps a primary key to a local row, for its own partition of each table. Local indexes are used by each host to find the tuples that it owns. Tigon maintains a primary index in the HWcc region (the *CXL index*) for tuples from the same table and partition that are resident in CXL memory. If a tuple is moved to CXL memory, the local row includes a pointer, `shortcut-ptr`, that points to the corresponding HWcc record, which in turn has the `SWcc-row-ptr` that points to the SWcc row. Otherwise, the `shortcut-`

ptr is NULL. Tigon pays the price of storing the shortcut-ptr in a local row to efficiently access a tuple moved to CXL memory without searching the CXL index. As tuples are moved between local DRAM and CXL memory, the owner host updates the CXL index while each non-owner host uses the CXL index to find tuples that it does not own.

Example. We use the example in Figure 2 to explain the data organization. Initially, the table is partitioned across the two hosts. Tuple (A, 6) belongs to the partition owned by Host 1 and is stored in the local DRAM of Host 1. So the shortcut-ptr of the local row is NULL. When transaction 1 from Host 1 needs to access tuple (C, 0), Host 2 moves this tuple to CXL memory by creating an HWcc record and an SWcc row, setting the SWcc-row-ptr of the HWcc record to point to the SWcc row, and inserting the HWcc record to the CXL index. It will also set the shortcut-ptr of the local row to point to the HWcc record. Host 1 can now get tuple (C, 9) using the CXL index.

Data movement policy. Tigon moves tuples to CXL memory upon request. If a host's HWcc memory usage exceeds a predefined threshold, it will move a tuple it owns from CXL memory back to local DRAM. Ideally, a host prioritizes moving tuples least likely to be accessed by non-owner hosts in the future back to local DRAM. This makes the Least Recently Used (LRU) policy a natural fit. Under LRU, the tuple least recently accessed by a non-owner host is moved back to local DRAM. However, maintaining the metadata required for LRU is expensive; LRU requires maintaining a linked list across tuples in CXL memory and updating this list for every access from a non-owner host.

To reduce overhead and minimize the HWcc memory usage, Tigon adopts the CLOCK policy [16]. Each tuple in CXL memory is associated with a bit (the clock-bit, as shown in Figure 3) stored in the HWcc record. The clock-bit is set whenever a non-owner host accesses the tuple. To move a tuple back, the owner host maintains a circular cursor to scan the tuples in CXL memory. If the clock-bit for a tuple is set, the host unsets it and moves the cursor forward. Otherwise, the host moves the tuple back to local DRAM.

Software cache coherence. We co-design a novel software cache coherence protocol that works with the database's internal synchronization. Tying coherence to synchronization has the advantage that the coherence granularity is large (i.e., a tuple, which is larger than a cacheline) and the maintenance of coherence metadata is less expensive because it is combined with synchronization tasks that are already part of the database's operation. The disadvantage is that the mechanism requires source code changes and is specific to the database, so it cannot be directly reused by other applications.

A database thread needs to take a latch before accessing a tuple to ensure only one thread can read or modify this tuple. Therefore, Tigon reuses this latch, HWcc-latch, to support software cache coherence by embedding the metadata, SWcc-bitmap, within the HWcc record.

The SWcc-bitmap uses a bit for each host (Tigon supports up to 16 hosts) to track the hosts that can do cacheable loads to SWcc rows, because cacheable loads have much lower latency than non-temporal loads. When a host reads an SWcc row, it looks at its bit in the SWcc-bitmap to determine how to access the data in the row. If the host's bit is set, it uses cacheable loads. Otherwise, its cached copies might be out of date. So it flushes the relevant cachelines, and reads the SWcc row from CXL memory into the CPU cache, and sets its bit in the SWcc-bitmap. So long as no other host updates the SWcc row (i.e., so long as the bit remains set), the host can use cacheable loads to read the SWcc row. A host that updates an SWcc row will unset the bits for all other hosts.

3.3 Transaction Execution

Transactions in Tigon use epoch-based group commit similar to SiloR [82]. Tigon breaks transactions into epochs and ensures that transactions in smaller epoch numbers are serially ordered and committed before those in larger ones.

Workers. Each host in Tigon includes multiple threads for executing transactions. A transaction is initially assigned to a thread, called a *transaction worker*, for executing and committing this transaction. The transaction worker may ask another host to do work (e.g., move data into CXL memory) to complete the transaction it is currently executing. We call the host the transaction worker runs on the *local host* and the partition owned by the local host the *local partition*. Similarly, we call the other hosts and the other partitions *remote hosts* and *remote partitions*, respectively, with respect to the transaction worker.

Read, write, and range query. Algorithm 1 shows the process of reading or writing a tuple in Tigon. If the tuple is in the local partition, the transaction worker gets the local row through the local index and further gets the HWcc record if the tuple resides in CXL memory (Lines 1-11 in Algorithm 1). To read or write a tuple (i.e., the ReadOrWrite function in Algorithm 1), the transaction worker checks the 2pl-lock to take a read or write lock or abort the transaction based on the 2PL protocol. For example, if the write lock bit of the 2pl-lock is already set, the transaction will be aborted. Otherwise, the transaction worker proceeds to perform read or write.

To read or write a tuple owned by a remote host (Lines 13-22 in Algorithm 1), the transaction worker will ask a remote host to move the tuple to CXL memory if it cannot find it in the CXL index. Note that we use a while loop to retry the data move in request because it is possible that the remote host moves the tuple back to local DRAM before the worker accesses this tuple. In practice, the worker will almost never retry due to the data movement policy of Tigon.

Processing range queries is similar to processing read operations. If a range query is on the local partition, the transaction worker searches the local index to get the local rows within the range and optionally follows the shortcut-ptr to access the tuples moved to CXL memory. Otherwise, the transaction

Algorithm 1: Read or write a tuple in Tigon.

```
Input: k – key of the tuple to be read or written
if k belongs to local partition then
    localRow ← Get(k, local index);
    Acquire local-latch for localRow;
    if shortcut-ptr of localRow is NULL then
        ReadOrWrite(localRow);
    else
        hwccRecord ← shortcut-ptr of localRow;
        Acquire HWcc-latch for hwccRecord;
        ReadOrWrite(hwccRecord);
        Release HWcc-latch for hwccRecord;
    Release local-latch for local row;
else
    hwccRecord ← Get(k, CXL index);
    while hwccRecord is NULL do
        Ask the owner host to move k to CXL memory;
        if the host replies k does not exist then
            return NOT_EXIST;
        else
            hwccRecord ← Get(k, CXL index);
    Acquire HWcc-latch for hwccRecord;
    ReadOrWrite(hwccRecord);
    Release HWcc-latch for hwccRecord;
```

worker will search the corresponding CXL index. If the CXL index does not include all tuples for the queried range, the worker asks the corresponding remote host to move all tuples in that range to CXL memory and searches the CXL index again.

Insert and delete. Inserting a tuple to local partition requires creating a local row for the tuple and inserting it into the local index. The challenge is inserting a tuple to a remote partition. To address this challenge, our key idea is to let the remote host prepare the index entry and metadata for the insert and let the transaction worker perform the actual insert operation. Specifically, the remote host is requested to create a local row with the `is-valid` flag set to False, insert the local row to its local index, and move the local row to CXL memory (the value of the `is-valid` flag is copied from the local row to the SWcc row). The `is-valid` flag prevents other transaction workers from reading or writing this tuple. Afterward, the transaction worker gets the HWcc record from the CXL index, acquires the write lock, completes the insert by copying the tuple to SWcc row and setting the `is-valid` flag to True.

The algorithm for deleting a tuple is similar to the one for reading or writing a tuple. If the tuple belongs to the local partition, the transaction worker acquires the write lock and sets the `is-valid` flag to False to delete the tuple. To delete a tuple from a remote partition, the transaction worker asks the remote host to move the tuple to CXL memory, gets the HWcc record from the CXL index, and sets the `is-valid` flag to False. Deleted tuples are reclaimed using epoch-based reclamation (details in 3.6).

3.4 Concurrency Control

Tigon manages concurrency at two levels: 1) maintaining and accessing the CAT under concurrent accesses across hosts; 2) providing serializability for concurrent transactions.

3.4.1 Maintaining and accessing the CAT

Tigon has the following synchronization goals for maintaining the CAT. (1) Even during data movement, HWcc records can be safely inserted and deleted; (2) a worker thread on the owner host can safely use the `shortcut-ptr`, even when tuples are moving between CXL memory and local DRAM; (3) workers from different hosts can concurrently and efficiently get HWcc records by caching and reusing the pointers to HWcc records within a transaction.

Tigon achieves these goals using (1) the `local-latch`, which can only be accessed by a thread on the owner host; (2) the CXL index, which can be accessed by any host; and (3) the `HWcc-latch`, which can be accessed by any host.

Data movement. In Tigon, only threads on the owner host can insert and delete HWcc records for data movement. If a non-owner thread wants to insert or delete a record, it sends a message to the owner host. The owner host will make inserted HWcc records visible to non-owner hosts by putting their entry in the CXL index, which it only does after all of its metadata structures are valid. Deleted HWcc records are removed from the CXL index by the owner host. HWcc records are moved from CXL memory to local DRAM by the owner host following a data movement policy.

Shortcut pointer. Owner hosts in Tigon can use the `shortcut-ptr` of the local row to find and update a tuple in CXL memory without searching the CXL index. Owner hosts can also move their own tuple into or out of CXL memory by inserting or deleting an HWcc record to/from the CXL index. The key challenge is to ensure that the value of the `shortcut-ptr` always reflects the latest state of an HWcc record, even when a worker thread accesses it while another thread wants to move it. The `shortcut-ptr` must either point to the correct HWcc record or it must be `NULL` if the tuple is not resident in CXL memory. Tigon solves this challenge by reusing the database latch for mutual exclusion. Each operation (i.e., data movement and tuple get) needs to take the `local-latch` of the local row before executing, ensuring that only one worker from the owner host can proceed at a time. Given that a non-owner host cannot move a tuple directly, the tuple get operation can then safely follow the `shortcut-ptr` to get the HWcc record. For data movement operations, they will then insert the newly created HWcc record into the CXL index, update the `shortcut-ptr` to point to the HWcc record, and release the `local-latch`.

Non-owner get. Non-owner hosts get an HWcc record through the CXL index and cache the pointer to the HWcc record for subsequent accesses within a transaction. When a tuple is being moved from CXL memory back to local DRAM, the non-owner host must be able to safely dereference the pointer to the HWcc record. At the start of a CXL memory to local DRAM move, Tigon marks the SWcc row as invalid by setting the `is-valid` flag to 0, allowing the non-owner host to know that if they read this entry, they have followed a stale pointer to an invalid entry. Tigon eventually reclaims invalid entries using epoch-based reclamation (details in §3.6), when it knows there are no more possibly stale pointers that point to these invalid entries.

3.4.2 Serializing transactions

Two-phase locking. We adapt 2PL to Tigon, as described in §3.3. 2PL requires a transaction to acquire either a read or write lock before accessing a tuple, depending on the type of the access. Lock acquisitions and releases are broken into two consecutive phases: the transaction only acquires or upgrades locks in the first phase and only releases or downgrades locks after all locks are acquired. Tigon uses Strong Strict 2PL (SS2PL), which acquires locks during transaction execution and releases them only upon completion [12, 24]. Tigon prevents deadlocks by aborting a transaction when a lock request is denied (NO_WAIT policy), following prior research [76] showing that this approach yields better scalability than deadlock detection for in-memory databases with a similar setup to Tigon.

As the first work to leverage CXL memory for building efficient distributed transactional databases, we choose to support conventional concurrency control protocols, 2PL and next-key locking, rather than optimistic concurrency control (OCC) and multi-version concurrency control (MVCC), even though OCC and MVCC may yield better performance for read-heavy workloads as shown in prior research [38, 68, 77, 79]. Supporting these protocols in Tigon requires addressing non-trivial technical challenges and goes beyond the scope of this paper.

Phantoms and next-key locking. Phantom problems [24] occur because 2PL locks only individual keys but does not lock the gaps between them. For instance, when a transaction scans a range, it locks only the tuples encountered during the scan, but allows other transactions to modify the range's membership (e.g., through inserts), violating serializability.

The classic solution to addressing phantom problems is *next-key locking* [55]. Records are accessed via an ordered index (such as a B$^+$-tree) and for inserts, deletes, scans, and point queries on non-existing keys, locks are acquired for both the key or key range and the *next* key in order.

We adapt next-key locking for Tigon. Supporting next-key locking for a local index is straightforward, but a challenge arises when a transaction worker needs to access data in a remote partition. In this case, the worker relies on the CXL index to perform next-key locking. Unfortunately, the CXL index may not include the next key as it only includes a subset of tuples from the local index.

To address this limitation, we enhance the CXL index with additional next-key information. Each tuple in the CXL index is augmented with the `has-next-key` flag indicating whether its next key in the CXL index is also the next key in the local index. With this mechanism, the transaction worker can safely determine whether it can directly lock the next key in the CXL index or needs to request a remote host to move the tuple for the next key to CXL memory.

The `has-next-key` information may be updated whenever an insert or delete operation is performed on the local index or a tuple is moved into or out of CXL memory. Additionally, the `has-next-key` information can be used to determine whether the CXL index has missing keys with respect to the local index for a given range, which can be used to support range queries on the CXL index.

3.5 Logging and Recovery

As mentioned in §3.1, to avoid 2PC, Tigon maintains the CAT to ensure each host can perform and log all modifications to tuples involved in a transaction. Our logging protocol is adapted from SiloR [82], a fast and scalable logging protocol for in-memory databases. Since SiloR is designed for an optimistic concurrency control protocol, we adapt its key ideas to design a logging protocol for 2PL in Tigon.

Key ideas. Tigon shares high-level ideas with SiloR [82], including epoch-based group commit and parallel value logging. Specifically, Tigon guarantees that transactions with a smaller epoch number are serially ordered and committed before those with a larger epoch number. Transactions of each epoch are committed by flushing the buffered log records to local storage. To support parallel logging, each worker thread independently generates a log record for each write at the commit phase, including the value of the tuple being modified and the epoch number and version number associated with the tuple. The log records are sent to dedicated logger threads for persistence. To support recovery, the protocol guarantees that the log record with the highest epoch and the highest version number within that epoch represents the most recent value for the tuple.

Logging. Tigon associates each transaction with an epoch number, and each tuple with an epoch number and a version number (i.e., `epoch-version` in Figure 3). The epoch number of a tuple indicates the epoch of the most recent transaction that modified it, while the version number specifies the most recent version within that epoch.

Tigon maintains a global epoch number, E, that advances periodically and is shared across all hosts via the HWcc region of CXL memory. Since E advances slowly (e.g., every 10 ms in our prototype), synchronizations on E will not be a bottleneck as SiloR shows [82].

During transaction commit, the transaction worker sets the epoch number of the transaction by reading the current

E from CXL memory. When applying writes to the tuple, the transaction worker updates the epoch number and version number as follows. If the transaction's epoch number equals to the tuple's, the version number is incremented. Otherwise, the tuple's epoch number is updated to the transaction's epoch number and the version number is reset to 0, indicating the start of a new epoch.

Log records. A log record is generated for each write, insert, and delete. It includes the identifier of the tuple (i.e., table ID and primary key), the `epoch-version`, and the corresponding value of the tuple or the delete operation. Each transaction worker stores the log records in a memory buffer and passes it to the logger thread when it fills or at an epoch boundary. Each logger thread reads log buffers and flushes them to local storage. In additional to the global epoch number, Tigon maintains a local epoch number, e_l, for each logger thread, in the HWcc region. Each logger thread increments e_l as it flushes the log buffers, indicating that all log records with an epoch number less than e_l are persisted to local storage. Transactions in an epoch e are regarded as committed if e is less than e_l of all logger threads.

Recovery. With this logging protocol, Tigon can adopt the same parallel recovery algorithm as SiloR. That is, Tigon can move all log records of committed epochs to CXL memory and then divide them across worker threads such that each thread applies assigned log records in parallel. For a log record that corresponds to a database tuple, it is applied to the tuple if it has a larger `epoch-version` than the last log record applied to this tuple. Otherwise, this log record is skipped.

3.6 Implementation

We implement Tigon in C++ by building on the Lotus [85] codebase, which has ∼18,000 LoC. We added ∼5,000 LoC. Tigon provides read, write, delete, insert, and range query APIs for developers to implement transactions or parameterized transactions. We implement local index and CXL index using an existing B$^+$-tree implementation [84], which adopts the optimistic crabbing index concurrency control protocol [42]. We extend the B$^+$-tree to support next-key locking. Tigon adopts offset pointers [14] to make CXL-resident data structures position-independent. CXL memory is exposed as a CPU-less NUMA node by Linux. We modify the mimalloc [41] memory allocator to use the CXL memory region.

Each host in Tigon includes multiple worker threads for processing transactions and data movement requests. If a transaction needs to access data of another host, its worker thread will send a data movement request using CXL memory as a transport. A dedicated input thread pulls these requests and dispatches them round robin to the local message queues of worker threads. The worker thread will process data movement requests from other hosts when it waits for the result of a data movement request it makes, or after a transaction is finished. If a transaction aborts due to a conflict, Tigon retries it until success.

We implement Tigon's CXL-based transport layer as lock-free multi-producer single-consumer (MPSC) ringbuffers in CXL memory. The metadata of the ringbuffer (e.g., head and tail) is stored in the HWcc region while the buffer entries are stored in the non-HWcc region. Each input thread is assigned a ringbuffer for receiving messages from other hosts.

Tigon uses epoch-based reclamation (EBR) [25] to ensure memory safety. To adapt EBR to the CXL pod, Tigon stores a local epoch number for each worker thread (in addition to the global epoch number and the per-logger local epoch numbers in §3.5) in the HWcc region and maintains per-worker retired objects list in the local DRAM of each host. Worker threads check the global epoch number periodically to determine the current epoch and update their local epoch number. Worker threads reclaim the memory objects retired in a given epoch only after all workers have exited that epoch.

4 Evaluation

We seek to answer the following questions:

- What are the end-to-end performance benefits of Tigon compared with the state-of-the-art? (§4.2)
- How does varying the hardware cache-coherent (HWcc) memory budget impact Tigon's performance? (§4.3)
- What is the performance benefit of software cache coherence compared with using HWcc memory only? (§4.4)
- How does logging impact the throughput and latency of Tigon? (§4.5)
- How effective are our optimizations? (§4.6)

4.1 Experimental Setup

CXL pod emulation. We emulate a CXL pod on a machine that includes an Intel Xeon Platinum 8568Y+ CPU, 512 GB local DRAM, and a 128 GB CXL 1.1 memory device. Both the local DRAM and CXL memory use DDR5 4800 DRAM. The CXL memory module has a single memory channel and is connected to the CPU via a PCIe 5.0 x8 link. This CXL memory device has a 1.6× higher latency than the local DRAM (259 ns vs. 159 ns), and its bandwidth is 13% of the bandwidth of the local DRAM (31.8 GB/s vs. 238.3 GB/s) using Intel's Memory Latency Checker [3] under a 3:1 read/write ratio. The machine is equipped with a Samsung Datacenter SSD [5] for persistent storage and a 100 Gbps Mellanox MT42822 BlueField-2 ConnectX-6 NIC.

We run 8 virtual machines (VMs) on this machine, each with 5 vCPUs and 10 GB local DRAM. We use SR-IOV to create the virtualized network for our VMs, using TCP over Ethernet. We configure the VMs to share the CXL memory device to emulate a CXL pod that includes 8 machines sharing CXL memory. There are currently no physical CXL devices that support inter-host cache coherence. For our setup, the inter-VM cache coherence is implemented by the physical machine's cache coherence, which will be faster than real inter-host hardware cache

coherence (when it becomes available). We analyze the potential impact of higher-latency inter-host cache coherence in §4.3.

We limit the HWcc region of the emulated CXL pod to 200 MB for all experiments. That is, our CXL memory allocator is configured to allocate no more than 200 MB of HWcc memory, despite the emulated CXL pod offering 128 GB. We study the impact of varying HWcc memory budgets in §4.3 and adjust the size of the HWcc region for that experiment only.

Baselines. We compare Tigon with three distributed databases, Sundial [77], DS2PL [85], and Motor [79]. We do not compare with Lotus [85] because it does not support all five transactions in TPC-C. We choose Sundial and DS2PL as representatives of traditional partition-based shared-nothing databases that support general-purpose transactional workloads, just as Tigon supports them. DS2PL uses two-phase locking (2PL) for reads and writes, similar to Tigon. Sundial includes an advanced concurrency control protocol that uses OCC for reads and 2PL for writes. Both Sundial and DS2PL are vulnerable to the phantom problem [24] that Tigon correctly addresses. Motor is a state-of-the-art distributed database that uses RDMA-based shared disaggregated memory and achieves persistence via replication.

Improved baselines. For a fairer comparison with Tigon, we enhance our chosen baselines in a number of ways. We implement next-key locking in DS2PL to avoid the phantom problem. We enhance Sundial and DS2PL to support durability using our logging protocol (§3.5). Sundial and DS2PL support only two of the five transactions in TPC-C, so we added implementations of insert, delete, and scan so they could execute all five.

Sundial and DS2PL originally use the network as their transport layer as they adopt the shared-nothing architecture, whereas Tigon uses a queue in CXL memory. To demonstrate that Tigon's superior performance is not simply due to its faster transport layer, we upgrade Sundial and DS2PL to also pass messages over CXL memory (*i.e.,* Sundial-CXL and DS2PL-CXL). Tigon uses 16 MB of CXL memory that is not hardware cache-coherent (non-HWcc) for its message buffers and we give the baselines 512 MB because they pass larger messages.

Using a memory queue for message passing allows us to make another improvement: we repurpose an I/O thread as a worker thread (for executing transactions). The IO thread was necessary because it wrote to a network device and could block in the OS, while our memory queue is non-blocking. We name these improved baselines Sundial+ and DS2PL+. We later show that these improved baselines outperform the original Sundial and DS2PL, and only compare Tigon with these improved baselines (§4.2).

Configurations and metrics. We run Tigon, Sundial, and DS2PL in the VMs of the emulated CXL pod. For each test, we run each system for 30 seconds to warm up and report the performance for the following 30 seconds. We mainly present throughput since Tigon prioritizes throughput over latency using group commit, a common practice widely adopted in transactional databases [9, 33, 69, 75, 77, 85]. We set each epoch to be 10 ms for the group commit protocol.

We run Motor on four machines, each including two Intel Xeon Platinum 8380 CPUs, 256 GB local DRAM, and a 25 Gbps Mellanox ConnectX-6 NIC. We use one machine as the compute server and three machines as memory servers, as the paper specifies [79]. Motor replicates its data in the three memory servers to achieve durability. We configure Motor to use 40 CPU cores to match the configuration of Tigon and the other two baselines.

Benchmarks. We use the full TPC-C [7] and a variant of YCSB [15] as our benchmarks. We configure TPC-C to use 24 warehouses (2.2 GB total), where each warehouse is a partition, and store 3 warehouses in each host (one warehouse per worker thread). We evaluate the standard mix for TPC-C's five stored procedures: 45% NewOrder, 43% Payment, 4% OrderStatus, 4% Delivery, and 4% StockLevel. We configure YCSB to generate a table including 2.4M tuples, each with one 4-byte key column and ten 100-byte value columns (2.2 GB total) [15]. The table is range-partitioned into 8 partitions across hosts. Each transaction performs 10 read or write operations with a configurable read/write ratio. The key accessed by each operation is generated using the Zipfian distribution with skewness factors of 0.7 and 0.99. We use 0.7 and 0.99 to simulate medium-contention and high-contention workloads, respectively, as done by prior research [79].

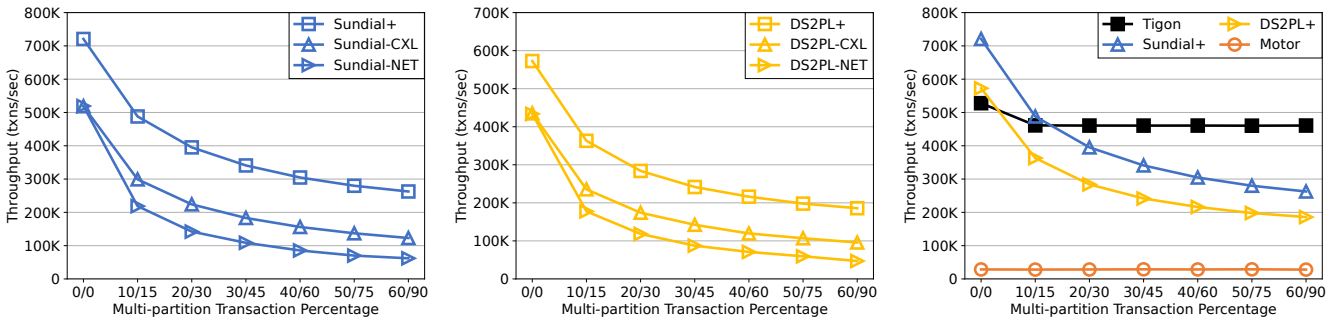
4.2 End-to-End Performance

We evaluate Tigon's end-to-end performance using TPC-C and YCSB.

TPC-C setup. The performance of the compared systems changes significantly based on how many transactions access data from multiple partitions. In the default configuration of TPC-C, 10% of NewOrder and 15% of Payment transactions access multiple warehouses (i.e., multi-partition transactions). Therefore, we vary the two percentages proportionally, up to 60% remote NewOrder and 90% remote Payment transactions (and we refer to this configuration as 60/90).

Performance of improved baselines. We measure the performance benefits of the improved baselines using TPC-C with varying percentages of multi-partition transactions. Figure 4a and 4b show that replacing the network with CXL memory increases the transaction throughput for the workloads that involve multi-partition transactions since these transactions need to exchange network messages during transaction execution and 2PC. For the 60/90 configuration, performance improves by 2.0× for both Sundial and DS2PL. Repurposing the I/O thread as a transaction thread brings the total improvement to 39% and 32% when there are no multi-partition transactions (0/0), and 4.2× and 3.9× at 60/90. The rest of our evaluation compares only against the optimized baselines.

TPC-C performance. Figure 4c shows TPC-C throughput under varying percentages of multi-partition transactions.



(a) Improved Sundial **(b)** Improved DS2PL **(c)** Tigon comparing with improved baselines

Figure 4: TPC-C performance comparison, varying percentages of multi-partition transactions. We optimize (a) Sundial and (b) DS2PL by replacing the original network transport layer (*-NET) with CXL (*-CXL), and again by replacing an I/O thread with a worker thread (Sundial+ and DS2PL+). (c) Then, we compare Tigon with Sundial+, DS2PL+, and Motor.

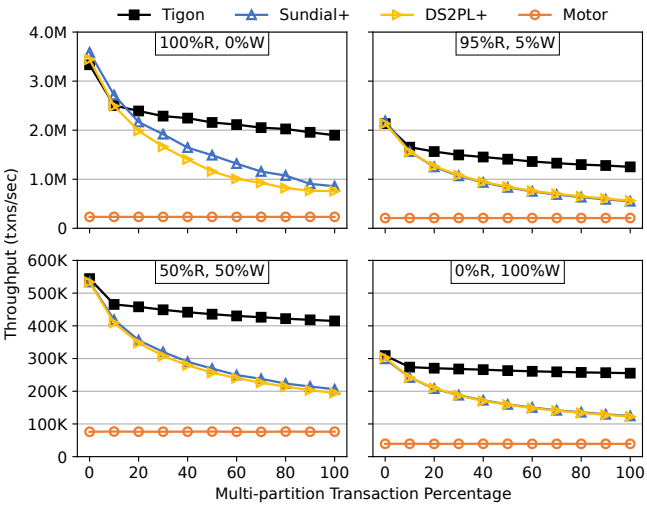


Figure 5: YCSB throughput, varying both read/write ratios and percentages of multi-partition transactions.

When there are no multi-partition transactions, Sundial+ and DS2PL+ are 37% and 8.5% faster than Tigon, respectively. The performance benefit of Sundial+ is due to its optimistic concurrency control protocol and the fact it does not avoid phantoms. Avoiding phantoms using next-key locking (§3.4.2) costs Tigon 10%-12% of its performance across all tested percentages of multi-partition transactions; for TPC-C's default configuration of 10/15, a variant of Tigon that allows phantoms (not shown in the figure) outperforms Sundial+ by 5.8%.

As the percentage of multi-partition transactions increases to 60/90, Tigon outperforms Sundial+ by 75% and DS2PL+ by 2.5×. The throughput of Sundial+ and DS2PL+ drops significantly due to the overhead of inter-host communication during transaction execution and 2PC. For instance, in the 60/90 case, Sundial+ and DS2PL+ send 3.3 and 4.1 messages per transaction, respectively, whereas Tigon moves the data that will be accessed across hosts to CXL memory (during the warmup phase) and does not send messages during transaction execution. More specifically, Tigon moves the `CUSTOMER` and `STOCK` tables to CXL memory, with `CUSTOMER` containing 720K tuples and `STOCK` containing 2.4M tuples. The two tables consume 176 MB of HWcc memory (for metadata) and 1.6 GB CXL memory in total. No data is moved from CXL memory to local DRAM during the experiment. For TPC-C, supporting software cache coherence allows Tigon to use 8.2× more CXL memory for sharing data.

Compared with Motor, Tigon achieves 15.9×-18.5× higher throughput mainly because Motor shares and replicates data in RDMA-based memory servers and suffers from the high latency of one-sided RDMA operations. We observe that the performance of Motor, approximately 30K/s, is constrained by the limited network bandwidth (25 Gbps) of the machines available to us. The original paper [79] reports that Motor achieves a maximum TPC-C throughput of approximately 100K/s with 24 warehouses and 40 CPU cores. Tigon uses the same configuration and its throughput ranges from 460K/s to 528K/s, as shown in Figure 4c.

YCSB setup. We run YCSB with a skewness factor of 0.7 under varying percentages of multi-partition transactions and different read/write ratios (i.e., 100R/0W, 95R/5W, 50R/50W, and 0R/100W). To generate a multi-partition transaction, we randomly choose a remote partition and have half (5) of the operations access that remote partition.

YCSB performance. Figure 5 shows the results. When there are no multi-partition transactions, the evaluated systems (except Motor) perform similarly (within 3.3%) for workloads with writes. For the read-only workload, Tigon outperforms both Sundial+ and DS2PL+ once the percentage of multi-partition transactions reaches 20% or higher. Comparing Tigon with Sundial+ at 100% multi-partition transactions, Tigon is 2.0×–2.3× faster. Tigon moves all 2.4M tuples to CXL memory as they are accessed across hosts. These tuples consume 112 MB of HWcc memory and 2.6 GB CXL memory in total. No Data is moved from CXL memory back to local DRAM, as HWcc memory usage remains well below the 200 MB limit.

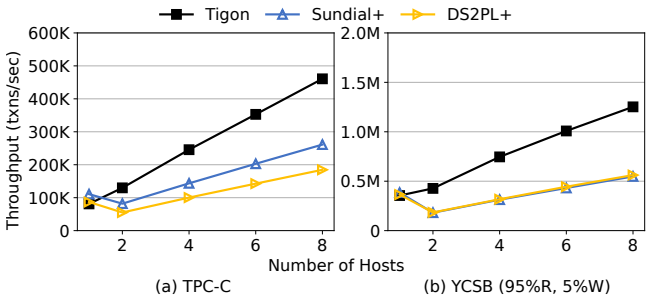


Figure 6: Throughput of TPC-C (60% remote NewOrder and 90% remote Payment) and YCSB (95%R, 5%W, 100% multi-partition transactions) on 1, 2, 4, 6, and 8 hosts.

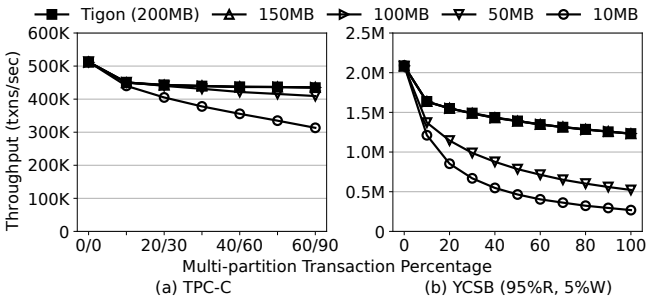


Figure 7: Throughput of TPC-C and YCSB (95%R, 5%W), varying both hardware cache-coherent memory budgets and percentages of multi-partition transactions.

Motor's performance is independent of the percentage of multi-partition transactions, but it is 5.4×–14.3× slower than Tigon. Results with a skewness factor of 0.99 show a similar trend. For example, with 100% multi-partition transactions, Tigon outperforms Sundial+ and DS2PL+ by 2.7× and 3.5× for the 50R/50W workload, respectively. We use a skewness factor of 0.7 for all remaining tests.

Scalability. We evaluate the scalability of Tigon, Sundial+, and DS2PL+ by running TPC-C with 60/90 multi-partition transactions and YCSB (95% reads, 5% writes) with 100% multi-partition transactions on configurations with 1, 2, 4, 6, and 8 hosts. As shown in Figure 6, Tigon scales efficiently, achieving a 5.7× throughput improvement for TPC-C and a 3.5× improvement for YCSB as the number of hosts increases from 1 to 8. In comparison, Sundial+ and DS2PL+ exhibit significantly lower scalability, with throughput gains of only 2.4× and 2.1× for TPC-C, and 1.4× and 1.5× for YCSB, respectively. The slope of Tigon's TPC-C scaling graph from 2 to 8 processors is 55.1Ktx/s per processor, while Sundial+ is 29.9Ktx/s per processor and DS2PL+ is 21.6Ktx/s per processor. For YCSB the slopes are 137.5Ktx/s per processor for Tigon, and 61.1Ktx/s per processor for Sundial+ and 63.0Ktx/s per processor for DS2PL+.

We are unable to empirically determine the scalability limit of Tigon due to hardware constraints that prevent large-scale experiments. Performance could plateau for many reasons: limited scalability of atomic instructions over CXL memory, especially under contention, limited scalability of hardware cache coherence, or the fixed size of the HWcc region could all become bottlenecks. Identifying and analyzing these limits is an important direction for future research.

4.3 Impact of Limited HWcc Memory

We measure TPC-C performance while varying the HWcc memory budget from 10 MB to 200 MB. Figure 7 shows the results. It is encouraging that even with as little as 50 MB of HWcc memory, Tigon performs only 5.8% slower than the configuration with unlimited HWcc memory.

With only 10 MB of HWcc memory, performance suffers because transactions wait for data to be moved into and out of CXL memory. When running TPC-C with a 60/90 mix of multi-partition transactions, 10 MB of HWcc memory requires Tigon to move 16K tuples in and out of CXL memory every second. Bandwidth is not a factor limiting performance; Tigon consumes only 367 MB/s of CXL bandwidth (approximately 1.1% of device bandwidth) in this configuration.

YCSB with a 95/5 read/write mix requires 100 MB to achieve full performance because there is more multi-partition sharing in YCSB than TPC-C—YCSB has only a single table and every data tuple has a probability of being remotely accessed. At 100% multi-partition transactions and with 10 MB HWcc memory, Tigon moves 110K tuples every second and consumes 2.9 GB/s of CXL bandwidth (approximately 9.1% of device bandwidth).

Inter-host cache coherence. We emulate the latency of inter-host cache coherence (i.e., back invalidations [2]) using inter-core cache coherence on our hardware prototype. This underestimates the cost of inter-host cache coherence for our evaluated systems. We cannot directly measure the number of back-invalidates our workloads would generate, because there are no hardware counters that report it and accurate modeling would require knowing the cache state. However, we can measure inter-host invalidates for our software cache coherence protocol (§3.2) for 60/90 TPC-C. We measure 12.0 million tuple accesses that require inter-host invalidations (which is 14.5% of total accesses, due to locality). If we conservatively estimate back-invalidations to be 4× slower than local invalidations, they would reduce performance by 41.4% (computed using the CXL miss latency times four). Tigon would be 2.8%, 45%, and 9.6× faster than Sundial+, DS2PL+, and Motor, respectively.

4.4 Software Cache Coherence

We evaluate the performance impact of software cache coherence by running TPC-C and YCSB (95% reads, 5% writes) in four configurations: only using HWcc memory (NoSWcc), using HWcc memory and only using non-temporal loads/stores to access CXL memory in the software cache-coherent (SWcc) region (NonTemporal), using HWcc memory and only allowing a

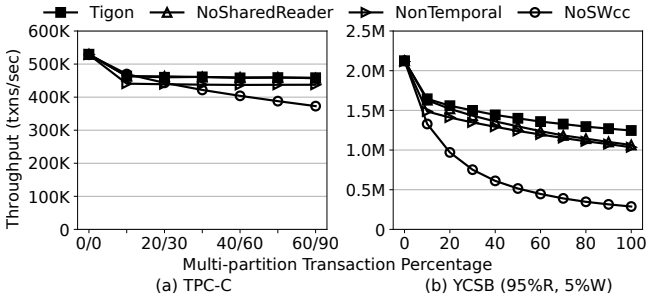


Figure 8: Throughput of TPC-C and YCSB (95%R, 5%W) with different software cache coherence protocols and varying percentages of multi-partition transactions.

	1ms	10 ms	20 ms	30 ms	40 ms	50 ms
TPC-C						
Tput (Ktx/s)	508	525	532	535	537	540
p50 (ms)	17.7	22.6	26.0	31.6	37.3	43.3
p99 (ms)	345.0	54.9	57.0	64.7	75.2	86.8
YCSB (50% read and 50% write)						
Tput (Ktx/s)	516	534	538	541	542	545
p50 (ms)	12.4	24.6	30.8	36.8	41.5	45.0
p99 (ms)	373.1	62.3	72.3	82.6	89.6	95.4

Table 1: Throughput, in Ktx/s (1,000 × transactions/second) and latency (in ms) for different epoch durations, running TPC-C and YCSB (50% reads, 50% writes) without multi-partition transactions.

single host to read data in the SWcc region (NoSharedReader), and Tigon with all optimizations for software cache coherence. NoSWcc avoids cache coherence problems by only using HWcc memory, but it suffers from the limited size of the HWcc region. NonTemporal provides cache coherence by avoiding the cache; accesses to the SWcc region use non-temporal loads and stores which bypass the CPU cache. NoSharedReader only allows reads of SWcc data for a single host at a time.

TPC-C. Figure 8 shows the benefits of Tigon's software cache coherence. When there are no multi-partition transactions, no data is moved to CXL memory and all variants perform identically. Across all tested percentages of multi-partition transactions, the NonTemporal configuration shows 4.5%-5.1% less performance than Tigon. The small performance win is because Tigon already caches tuples in local DRAM, which obviates the performance advantage of cacheable access to CXL memory. NoSWcc performs well at low multi-partition transaction percentages because it uses the HWcc region to store data; this data can be cached, leading to higher performance. However, as the number of multi-partition transactions increases, the performance of NoSWcc drops because it requires frequent data movement. At 60/90, NoSWcc is 19% slower than Tigon. The loss of performance comes from the latency of moving 9K tuples in and out of CXL memory every second. The workload consumes 641 MB/s of CXL bandwdith, well within the device's capabilities. TPC-C does not have read-shared data, so NoSharedReader has almost identical performance with Tigon.

YCSB. YCSB has significant read-sharing and shared data. NoSWcc is 19% slower than Tigon for 10% multi-partition transactions, but it is 4.3× slower (and the lowest performing option) at 100% multi-partition transactions. At 100% multi-partition transactions NoSWcc's performance is limited by the latency of data movement to the HWcc region—115K tuples/s (but only 3.0 GB/s bandwidth use; ∼9.4% of CXL bandwidth). Across all tested percentages of multi-partition transactions, Tigon is 11%-20% faster than NonTemporal because data is repeatedly accessed and Tigon can cache the tuples, which is much faster than non-temporal reads. But the advantage is small because Tigon caches tuples in local DRAM. At 100% multi-parition transactions, NoSharedReader is 15% slower than Tigon. NoSharedReader's performance suffers because it manipulates the `SWcc-bitmap`, and the read-sharing of data in YCSB means that NoSharedReader incurs 2.6× more cacheline flushes than Tigon.

4.5 Logging and Latency

Tigon uses epoch-based group commit (based on SiloR [82]) to ensure that it can maintain high throughput at reasonable latency. Committing logs at end of each transaction would provide low latency, but the resulting small, synchronous writes would severely under-utilize storage bandwidth. The use of virtualization further drops the storage bandwidth realized inside the VMs (from 600 MB/s to 380 MB/s, tested using `dd` with 4MB block size). To better utilize storage performance, Tigon collects 4 MB of logs before flushing them to SSD.

Table 1 shows the throughput and latency for a variety of logging epochs, running TPC-C and YCSB (50% reads, 50% writes) with no multi-partition transactions. For a 10 ms epoch, when compared with a 50 ms epoch, throughput for TPC-C drops only 2.8%, while its p50 latency drops by 48%. With a 10 ms epoch, Tigon has only 6.0% lower throughput for TPC-C compared with Tigon without logging. YCSB shows a similar result. Since Sundial+ and DS2PL+ adopt the same logging protocol, their latencies are similar to Tigon. Motor does not do group commit, but instead uses three in-memory replicas to achieve durability, so their p50 latencies are in the hundreds of microseconds. Tigon, instead, prioritizes throughput over latency using group commit.

4.6 Optimizations

Using CLOCK. Tigon moves data into CXL memory on demand, and uses the CLOCK algorithm to determine which data to move back to local DRAM (§3.2). Tigon uses CLOCK to minimize its use of HWcc memory and reduce the overhead of maintaining the replacement metadata; we compare its performance to the more accurate LRU policy to quantify the accuracy sacrificed for metadata efficiency.

Implementing LRU requires more CXL memory and more synchronization (than CLOCK) to maintain the LRU list. If we place the LRU list pointers in HWcc memory, YCSB will require 149 MB of HWcc memory to move all its tuples into CXL memory, which is 33% more than CLOCK (112 MB). If we limit the HWcc memory budget to 100 MB, LRU is 2.4× slower than CLOCK for YCSB with a 95/5 read/write mix and 100% multi-partition transactions because HWcc memory is oversubscribed. If we set the HWcc memory budget to unlimited and run the same experiment, LRU is 17% slower than CLOCK because of lock contention while manipulating the LRU list. CLOCK is therefore preferred due to its metadata efficiency.

Shortcut pointer. Tigon maintains the `shortcut-ptr` in the local row (Figure 3) to allow the owner host to find the tuple in CXL memory without looking through the CXL index. Across varying percentages of multi-partition transactions (excluding 0%), shortcut pointers improve throughput by 16% for TPC-C, and by 8.3%–24% for YCSB with 95% reads and 5% writes.

Tracking tuple modifications. Tigon allows the owner host to read CXL-resident tuples from local DRAM rather than CXL memory when the tuple is clean (i.e., the `is-dirty` flag in Figure 3 is False). This optimization is important for read-heavy workloads; YCSB read-only throughput improves by 60% for 10% multi-partition transactions and by 27% for 100% multi-partition transactions.

5 Related Work

CXL-enabled memory tiering. A lot of recent work on CXL memory management focuses on system software moving memory pages between local and CXL memory on behalf of applications [22, 54, 59, 65, 83]. Much of the work focuses on accurate detection of hot pages and policies for moving pages. Some hot pages only contain a few hot lines [63], making it difficult to recoup the cost of migration and motivating systems where hardware migrates cachelines [83].

CXL-enabled memory pooling and sharing. Recent work explores how CXL-enabled memory pooling and sharing can reduce cost in datacenters and HPC systems [11, 28, 45, 70], benefit serverless computing [28, 57], and accelerate distributed applications [50, 51, 81]. Distributed applications sharing CXL memory are vulnerable to partial failures [81, 86]. CXL-SHM [81] describes a distributed memory management system based on reference counting that tolerates partial failures.

Databases over CXL memory. Many papers consider optimizing databases using CXL memory [8, 10, 39, 60] such as elastically allocating CXL memory [39] or leveraging it for data shuffling [10]. One paper considers mapping SSD to the memory address space through CXL protocols and adopting hardware-based optimizations (e.g., (de)compression) when databases interact with this memory address space [40]. Two papers discuss the research opportunities of building databases over CXL memory [27, 43], but neither considers building a distributed transactional database over CXL memory. Pasha [29] describes a database architecture for CXL pods, but Tigon is the first system designed, implemented, and evaluated for a pod.

Distributed databases. Traditional distributed databases adopt a partition-based shared-nothing architecture, but their performance degrades quickly as the number of multi-partition transactions increases [17, 33, 56, 66]. Many papers optimize this architecture by reducing the number of multi-partition transactions [18, 49, 58, 61], optimizing concurrency control protocols [37, 52, 77, 85], or eliminating or reducing the cost of 2PC [23, 34, 46, 48, 67]. Tigon's performance remains robust because it uses CXL memory to synchronize multi-partition transactions instead of relying on message passing.

Many papers study RDMA-based distributed transactional databases [4, 13, 19–21, 32, 44, 62, 71–73, 75, 78–80, 87]. Some of them optimize partition-based databases [13, 20, 21, 32, 62, 71–73] while more recent papers adopt a shared disaggregated memory architecture to avoid multi-partition transactions [4, 19, 44, 75, 78–80, 87]. Motor [79] is a recent instance of this design and our evaluation shows the limits of its performance due to the high cost of RDMA relative to CXL memory access (§4).

6 Conclusion

This paper introduces Tigon, the first distributed in-memory database that synchronizes cross-host concurrent data accesses via atomic operations on CXL memory. Tigon addresses the inherent limitations of CXL memory, including higher latency, lower bandwidth, and limited hardware support for cache coherence, through three key innovations: a novel software cache coherence protocol for extending the cache-coherent region, efficient concurrency control protocols for maintaining and accessing data in CXL memory, and a scalable logging protocol for avoiding two-phase commit. Tigon is publicly available at `https://github.com/ut-datasys/tigon`.

7 Acknowledgment

We thank Nam Sung Kim at UIUC for technical consultation and for providing an experimental testbed. We thank our shepherd, Phillip Levis, and the anonymous reviewers for their constructive feedback. We thank Zixuan Wang at SJTU for his help with RDMA setup. Our work is supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We used the Chameleon testbed [35] supported by the National Science Foundation for development and artifact evaluation. Vijay Chidambaram was partially supported by Toyota during this project.

References

[1] CockroachDB Serverless. https://www.cockroachlabs.com/blog/announcing-cockroachdb-serverless/. (Accessed: May 2025).

[2] Compute Express Link (CXL) Specification, Revision 3.2. https://computeexpresslink.org/wp-content/uploads/2024/11/CXL-Specification_rev3p2_ver1p0_2024October2_evalcopy.pdf. (Accessed May 2025).

[3] Intel® Memory Latency Checker v3.11b. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html. (Accessed May 2025).

[4] Oracle RAC. https://www.oracle.com/technetwork/database/options/clustering/overview/new-generation-oracle-rac-5975370.pdf. (Accessed: May 2025).

[5] Samsung MZQL2960HCJR-00A07. https://semiconductor.samsung.com/us/ssd/datacenter-ssd/pm9a3/mzql2960hcjr-00a07/. (Accessed: May 2025).

[6] SK hynix Presents CXL Memory Solutions Set to Power the AI Era at CXL DevCon 2024. https://news.skhynix.com/sk-hynix-presents-ai-memory-solutions-at-cxl-devcon-2024/. (Accessed May 2025).

[7] TPC Benchmark C. https://www.tpc.org/tpcc/. (Accessed May 2025).

[8] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna T. Malladi, and Yang-Seok Ki. Enabling CXL memory expansion for in-memory database management systems. In Spyros Blanas and Norman May, editors, *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*, pages 8:1–8:5. ACM, 2022.

[9] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The new SQL server in the cloud. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1743–1756. ACM, 2019.

[10] Alexander Baumstark, Marcus Paradies, Kai-Uwe Sattler, Steffen Kläbe, and Stephan Baumann. So far and yet so near - accelerating distributed joins with CXL. In Carsten Binnig and Nesime Tatbul, editors, *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, pages 7:1–7:9. ACM, 2024.

[11] Daniel S. Berger, Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Mark D. Hill, and Rodrigo Fonseca. Octopus: Scalable Low-Cost CXL Memory Pooling. https://arxiv.org/pdf/2501.09020, 2025. (Accessed May 2025).

[12] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[13] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[14] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 191–203, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A PAGING EXPERIMENT WITH THE MULTICS SYSTEM*. Project MAC. Massachusetts Institute of Technology, 1968.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[18] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1):48–57, 2010.

[19] Alex Depoutovitch, Chong Chen, Per-Åke Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. Taurus MM: bringing multi-master to the cloud. *Proc. VLDB Endow.*, 16(12):3488–3500, 2023.

[20] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[21] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[22] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and general datacenter transactions for on-disk databases. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 343–360. USENIX Association, 2023.

[24] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[25] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[26] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[27] Yunyan Guo and Guoliang Li. A cxl-powered database system: Opportunities and challenges. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5593–5604. IEEE, 2024.

[28] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and YongWei Wu. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 421–437, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. Pasha: An efficient, scalable database architecture for cxl pods. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org, 2025.

[30] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory sharing with cxl: Hardware and software design approaches, 2024.

[31] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1504–1517, 2024.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

[33] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[34] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 145–161. ACM, 2021.

[35] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[36] Kimberly Keeton. The machine: An architecture for memory-centric computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Ziliang Lai, Hua Fan, Wenchao Zhou, Zhanfeng Ma, Xiang Peng, Feifei Li, and Eric Lo. Knock out 2pc with practicality intact: a high-performance and general distributed transaction protocol. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 2317–2331. IEEE, 2023.

[38] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.

[39] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. Elastic use of far memory for in-memory database management systems. In Norman May and Nesime Tatbul, editors, *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*, pages 35–43. ACM, 2023.

[40] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. Database kernels: Seamless integration of database systems and fast storage via cxl. In *CIDR*, 2024.

[41] Daan Leijen, Benjamin Zorn, and Leonardo De Moura. *Mimalloc: Free List Sharding in Action*, volume 11893 of *Lecture Notes in Computer Science*, page 244–265. Springer International Publishing, Cham, 2019.

[42] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.

[43] Alberto Lerner and Gustavo Alonso. CXL and the return of scale-up database engines. *CoRR*, abs/2401.01150, 2024.

[44] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. Gaussdb: A cloud-native multi-primary database with compute-memory-storage disaggregation. *Proc. VLDB Endow.*, 17(12):3786–3798, August 2024.

[45] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, BC Canada, March 2023.

[46] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[47] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1203–1217, New York, NY, USA, 2025. Association for Computing Machinery.

[48] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based commit and replication in distributed OLTP databases. *Proc. VLDB Endow.*, 14(5):743–756, 2021.

[49] Yi Lu, Xiangyao Yu, and Samuel Madden. STAR: scaling transactions through asymmetric replication. *Proc. VLDB Endow.*, 12(11):1316–1329, 2019.

[50] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, and Tao Ma. HydraRPC: RPC in the CXL era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 387–395, Santa Clara, CA, July 2024. USENIX Association.

[51] Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson. Telepathic datacenters: Fast rpcs using shared cxl memory, 2024.

[52] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, 2014.

[53] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[54] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[55] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 392–405. Morgan Kaufmann, 1990.

[56] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[57] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. Āpta: Fault-tolerant object-granular cxl disaggregated memory for accelerating faas. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–215, 2023.

[58] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 61–72. ACM, 2012.

[59] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Mtm: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA, 2024. Association for Computing Machinery.

[60] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. A three-tier buffer manager integrating CXL device memory for database systems. In *40th International Conference on Data Engineering, ICDE 2024 - Workshops, Utrecht, Netherlands, May 13-16, 2024*, pages 395–401. IEEE, 2024.

[61] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, 2016.

[62] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[63] Yan Sun, Jongyul Kim, Douglas Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems, November 2024.

[64] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Bijan Tabatabai, James Sorenson, and Michael M. Swift. FBMM: Making memory management extensible with filesystems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 785–798, Santa Clara, CA, July 2024. USENIX Association.

[66] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed SQL database. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online*

conference [Portland, OR, USA], June 14-19, 2020, pages 1493–1509. ACM, 2020.

[67] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM, 2012.

[68] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[69] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052. ACM, 2017.

[70] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. Evaluating emerging cxl-enabled memory pooling for hpc systems. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, November 2022.

[71] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[72] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 335–347. USENIX Association, 2017.

[73] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[74] Emmett Witchel. Challenges and opportunities for systems using CXL memory. In *ASPLOS 2024*, 2024.

[75] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. Polardb-mp: A multi-primary cloud-native database via disaggregated shared memory. In Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan, editors, *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, pages 295–308. ACM, 2024.

[76] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, 2014.

[77] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, 2018.

[78] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transaction can scale. *Proc. VLDB Endow.*, 10(6):685–696, 2017.

[79] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling Multi-Versioning for distributed transactions on disaggregated memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 801–819, Santa Clara, CA, July 2024. USENIX Association.

[80] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: fast one-sided rdma-based distributed transactions for disaggregated persistent memory. In Dean Hildebrand and Donald E. Porter, editors, *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 51–68. USENIX Association, 2022.

[81] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 658–674, New York, NY, USA, 2023. Association for Computing Machinery.

[82] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 465–477, USA, 2014. USENIX Association.

[83] Yuhong Zhong, Daniel S. Berger, Carl A. Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 37–56. USENIX Association, 2024.

[84] Xinjing Zhou. BTree. https://github.com/zxjcarrot/2-Tree/tree/master/backend/btreeolc. (Accessed: May 2025).

[85] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. Lotus: Scalable multi-partition transactions on single-threaded partitioned databases. *Proc. VLDB Endow.*, 15(11):2939–2952, 2022.

[86] Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. Lupin: Tolerating partial failures in a cxl pod. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, DIMES '24, page 41–50, New York, NY, USA, 2024. Association for Computing Machinery.

[87] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. Is scalable OLTP in the cloud a solved problem? In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023.

Mako: Speculative Distributed Transactions with Geo-Replication

Weihai Shen[†] Yang Cui[§] Siddhartha Sen[‡] Sebastian Angel[⋆] Shuai Mu[†]
[†]*Stony Brook University,* [§]*Google,* [‡]*Microsoft Research,* [⋆]*University of Pennsylvania*

Abstract

This paper introduces Mako, a highly available, high-throughput, and horizontally scalable transactional key-value store. Mako performs strongly consistent geo-replication to maintain availability despite entire datacenter failures, uses multi-core machines for fast serializable transaction processing, and shards data to scale out. To achieve these properties, especially to overcome the overheads of distributed transactions in geo-replicated settings, Mako decouples transaction execution and replication. This enables Mako to run transactions speculatively and very fast, and replicate transactions in the background to make them fault-tolerant. The key innovation in Mako is the use of *two-phase commit* (2PC) speculatively to allow distributed transactions to proceed without having to wait for their decisions to be replicated, while also preventing unbounded cascading aborts if shards fail prior to the end of replication. Our experimental evaluation on Azure shows that Mako processes 3.66M TPC-C transactions per second when data is split across 10 shards, each of which runs with 24 threads. This is an 8.6× higher throughput than state-of-the-art systems optimized for geo-replication.

1 Introduction

Highly available and reliable serializable transactional storage systems (e.g., Google's Spanner [20]) are the foundation of many Internet services. To achieve high availability despite datacenter failures, data is replicated across datacenters [7, 20, 30, 98]. To support ever-increasing amounts of data and request volumes, data is partitioned into many shards. A distributed transaction is used to guarantee consistency whenever the application needs to access data across shards.

The main drawback of these systems is that the coordination required by distributed transactions introduces great inefficiencies, to the point that the throughput that one can achieve with a single in-memory multi-core transactional key-value store is thousands of times higher than what distributed transactional systems typically provide [50, 55, 59, 60, 95]. At the same time, we cannot just replace a distributed transactional system with a fast single-machine variant because of the need for fault-tolerance and scalability.

Since an obvious bottleneck in distributed transactions is that the network latency is orders of magnitude higher than the CPU-memory latency, a trend in accelerating distributed transactions is to adopt ultra-low latency networking technologies such as RDMA and smart NICs [11, 17, 25, 46, 74, 90, 91, 106]. However, these techniques are most effective within a datacenter. In geo-replicated systems, the benefits of kernel bypass and faster local networks are negligible compared to the long latencies that are fundamental to wide-area networking.

Given the above, in this work we ask the question: *can the throughput achieved by distributed transactions in geo-replicated settings approach that of transactions within a single machine?* Answering this question in the affirmative has the potential to reduce the amount of machines that existing storage systems need to meet their current demands, while also empowering new applications that require high-throughput strongly consistent geo-replicated transactional storage (e.g., Nimble [5], Zanzibar [75]). But to realize this promise, we need to fundamentally rethink how we design distributed transaction protocols in the first place.

After a close examination, we conclude that a key limitation of current designs of distributed transaction protocols is the *tight* dependence between the two main building blocks: the *transaction coordination* and the *replication* protocols. Consider for example Spanner [20], which uses *two-phase commit* (2PC) as its coordination protocol to perform distributed transactions. Since 2PC is not fault-tolerant, Spanner synchronously geo-replicates all of the decisions made by 2PC participants (storage shards) after every step of the protocol. This ensures that failures are recoverable.

Several distributed transaction systems such as Tapir [109], Ocean Vista [30], and Janus [69] argue that transaction coordination and replication should be coalesced into a single protocol (an even tighter coupling) in order to avoid paying for common functionality twice. We argue that the opposite should be the case in WAN settings: that transaction coordination and replication should be decoupled even further! Our observation is that fully decoupling these two components allows the system to perform work speculatively, thereby masking the high overhead introduced by geo-replication.

To explore the benefits and limitations of this idea, we build *Mako*.[1] Mako is the first transactional sharded key-value store that hides the cost of geo-replication by speculatively processing distributed transactions. Of course, the idea of using speculation in transactional databases is not new. Prior to Mako, works like Rolis [85] and Amazon Aurora [98] leverage speculation to speed up transaction processing. But Rolis only has a single shard (and therefore cannot scale to

[1] Mako sharks are known for migrating long distances.

large databases), while Aurora does not support transactions across shards. The fundamental reason for this is that in an ideal world, when one performs a speculative cross-shard transaction, as soon as the transaction finishes executing one wants to start processing the next transaction without having to wait for the geo-replication to complete (otherwise we are back to square one where the geo-replication is the bottleneck for transaction processing). But if one starts executing other transactions that depend on values written by a speculative transaction before it has been geo-replicated, then the system can end up in a situation where a speculative transaction fails and then all transactions that depend on it must abort (and the transactions that depend on the aborting transactions also must abort, etc.). Such a *cascade of aborts* is troubling in part because in a concurrent and asynchronous system it is hard to efficiently track *which transactions depend on the failed transaction*. Without enough information, the cascade could become unbounded [28, 36, 79], and require pausing the system and aborting all pending transactions.

Mako's key technical contribution is precisely to introduce an architecture that allows it to safely speculate without having to wait for the results of geo-replication, while keeping track of very little information that is efficient to collect and yet sufficient to avoid unbounded cascading aborts.

Overview of Mako. Mako first executes and certifies the transactions among shard leaders. Shard leaders are in-memory multi-core stores that run a fast concurrency control protocol tailored for single-machine multi-core databases [85]. If some (or all) of these leaders are in the same datacenter, Mako uses a fast networking stack (DPDK) to connect them. At this stage, there is no replication so all transactions have been executed speculatively and the clients have not yet received a response. In the background, Mako performs parallel state-machine geo-replication across datacenters while also continuing to speculatively execute new transactions. On successful replication, the multi-core follower replicas will deterministically replay the execution results of the leader shards. Should something happen that prevents the replication from completing, Mako rolls back the speculatively executed transactions and makes sure that all live replicas are in a consistent state [38, 76] before moving on. Clients are notified after the replication completes.

For transaction coordination, Mako uses *two-phase commit* (2PC) speculatively. This means that if a shard leader fails before the 2PC result is replicated, the transaction's result becomes unrecoverable and the transaction must abort. To avoid unbounded cascading aborts, Mako uses a *distributed vector clock* (§4.2) as a coarse-grained dependency tracking method, and combines it with a *vectorized watermark* technique to selectively roll back only the affected transactions (§5.2).

Our evaluation of Mako on Azure as well as several state-of-the-art baselines show that Mako's throughput is the highest among all in the setting of geo-replication by at least 8.6× (at 10 shards) while introducing little additional latency. When we consider a setting without geo-replication (a single datacenter, which is not the setting for which we designed Mako), Mako's throughput is 50% lower than prior RDMA-based systems that tightly couple replication and concurrency control.

2 Background and motivation

In this paper, we study sharded and geo-replicated in-memory key-value stores that support transactions at very high throughput (close to a single-machine multi-core database) and with strong isolation (serializability). To the best of our knowledge, no existing data store achieves all of these goals. We discuss why this is the case next.

2.1 Limitations of existing designs

We take a closer look at existing designs to see where the core issues arise. One thing we find is that systems like Spanner [20] and FaRM [25] follow a classic approach: overlay the distributed transaction protocol on top of the replication protocol. That is, they use replication to checkpoint the key steps in transaction execution and commit.

The workflow of these systems is roughly as follows. A transaction first executes: it accesses data from different shards. During the execution, the system applies locking or optimistic concurrency control to avoid inconsistent accesses. At the end of execution, the system tries to certify (commit) the transaction. This is done via a two-phase commit (2PC) across all participating shards. To make the system fault-tolerant, the system applies replication (e.g., primary-backup in FaRM; Paxos [13, 21] in Spanner) to each critical step. For example, Spanner replicates the prepare phase and the commit phase in 2PC. Since this is expensive, prior works [30, 69, 109] optimize the geo-latency by mixing the replication and transaction protocols. But overall, the transaction certification decision is not known until replication finishes.

By looking at this design we can make two observations. First, it explains why distributed transactions can be a thousand times slower than multi-core transactions: the transaction execution and 2PC involve network messages, whose latency is a thousand times higher than the CPU-memory latency. In recent years, with the deployment of advanced networking hardware and kernel bypassing techniques, this issue can be mitigated within a datacenter. For example, FaRM [25, 26] equipped with RDMA can reduce the remote access latency and achieve very high per-shard throughput.

However, our second observation is that this problem cannot be addressed by advanced hardware in the setting of geo-replication, because the latency across datacenters will nevertheless remain high. In this case, the latency introduced by replication will result in a distributed transaction requiring tens (hundreds, or even thousands) of milliseconds to finish the certification. If we stick to the conventional architecture, there is no obvious way to address this issue.

One might argue that we can perhaps address the problem by increasing the number of concurrent requests to hopefully saturate the system's throughput. Normally, even though the latency is high, a system can still achieve high throughput if it can process many concurrent requests. This is unfortunately not true in the case of transactional systems. The throughput of transactional systems is sensitive to the contention in its workload. In most cases, increasing the number of concurrent requests increases contention too. For example, many Amazon customers buying the same items. As has been noted in a lot of prior work [42,68,82,105,109] and our own experiments (§7.6), more concurrency actually lowers throughput.

3 Challenges and key ideas

Given that in geo-replication the high latency of the transaction certification will inevitably decrease throughput, one way forward is to remove replication from the critical path of transaction coordination [78,94]. One example of a prior work that follows this principle is Calvin [94], which pre-determines the order of transactions into a totally ordered log, and then ships this log to remote replicas. The challenge with a design like Calvin's is that if the servers are fast multi-core machines, it is really hard for the log replay to be as fast as a multi-core storage server (e.g., Silo [95]) that processes the transactions without any pre-determined order. Therefore, the system's throughput is limited. We demonstrate this empirically in our evaluation (§7). Mako avoids this issue by decoupling transaction coordination and replication in a different way.

In a nutshell, Mako works as follows. A transaction executes and uses 2PC to certify the transaction among the shard leaders. No replication takes place during the execution and 2PC. Since the leaders accessed by a transaction are very often strategically deployed in the same datacenter [18,20,83,86,89], we use kernel bypass network acceleration (DPDK) to reduce the intra-datacenter latency. When Mako successfully certifies a transaction, the transaction's writes become speculatively visible to later transactions, and its log entry is replicated in the background to other datacenters. In the foreground, Mako goes on to execute the next transaction speculatively without being blocked by the replication. Note that *Mako does not return the results to the client until the replication finishes*. In this way, Mako can increase the throughput of the system, while maintaining a similar latency as prior works that provide strong consistency.

Challenges. We face three main challenges.

Challenge 1: How to speculate with 2PC in the presence of failures? A problem with speculative execution is that roll backs are required when speculation fails. For example, if there was a problem replicating the transaction's result, we may need to revert the transaction. Mako faces the extra challenge that the speculation spans multiple shards. Consider the following example. Suppose that we speculatively execute transaction T_0 over shards S_0 and S_1, then speculatively execute transaction T_1 (which depends on T_0) over shards S_1 and S_2, and finally speculatively execute transaction T_2 (which depends on T_1) over shards S_2 and S_3. Suppose that before T_0 finishes its replication protocol, shard S_0 fails. If we are using 2PC for the coordination (as is the case in Mako), losing a participant in 2PC means the transaction is now unrecoverable and must be aborted. But notice that even though T_1 and T_2 operate on shards that did not fail and their replications may have even succeeded, because they transitively depend on T_0—which is unrecoverable—they too must abort.

The crux of the issue is that 2PC does not interact well with speculation and failures. One may think that we could instead use a fault-tolerant version of 2PC, namely *three phase commit* (3PC) [87], but this is also not the case. In the above example, if shards S_0 and S_1 fail, transaction T_0 cannot be recovered since 3PC cannot handle *all* of its participants failing. And while T_1 can recover from S_1's failure, it depends on T_0, so it too must abort. Finally, T_2 had nothing to do with any of the failing shards but it also must abort because it depends on failed transactions. In other words: the *per-transaction fault tolerance* of 3PC [49] does not extend across transactions, and hence it does not preclude cascading aborts.

Challenge 2: How can we avoid the overhead of serializing transaction results into a sequential log?[2] This is the main challenge in achieving high throughput. One reason that multi-core transactions are fast is that they *avoid* generating a global serialization order among all transactions. If we adopt the traditional approach of using one log per shard (e.g., Spanner), this will limit performance [8,15,19,20].

Challenge 3: How can we sequence conflicting log entries from different logs? As different cores may access the same data, they could write conflicting entries to different logs. When followers replay these logs, how do we guarantee that the replay will lead to the same state as the leader?

Key ideas. Mako cannot prevent cascading aborts. What Mako does, however, is to bound the impact of failed speculation. To address challenges 1 and 3 while retaining good performance, Mako introduces a distributed vector clock as a coarse-grained dependency tracking method, and a vectorized watermark to selectively rollback affected transactions, while allowing uninvolved transactions to continue executing. To address challenge 2, Mako uses one log per core and ideas from prior work [85], but data is *not* sharded across cores (multiple cores may access the same data).

4 Design of Mako

In this section, we will describe the main ideas of Mako. We first give an overview and then provide the details of how Mako processes transactions.

[2] Serialization here refers to ordering concurrent requests into a sequence, not converting a data structure into a byte stream.

4.1 Overview

Figure 1 shows the architecture of Mako. A Mako deployment splits its data (key-value pairs) into multiple shards. Each shard is replicated to multiple datacenters with a leader-follower architecture. Similarly to Spanner [20] and other geo-replicated databases, the leaders of different shards ("shard leaders") in Mako can be in different datacenters, but it is advantageous for performance reasons to ensure that the shards that are often accessed together in transactions have their leaders co-located in the same datacenter. In practice, many workloads show such a strong data locality [14, 18, 53, 96]. For example, in a car sharing application such as Uber or Lyft, each ride could be expressed as a transaction between the user, the driver, and the start and destination. The data of all of these entities could be sharded and distributed according to geographic locations [96]. Most transactions will happen locally within a city and hence within the same datacenter, while a few transactions will cross datacenters (e.g., travel from one city to another).

A transaction enters Mako and goes through the following phases before it is finished.

Execution: A shard leader serves as the *coordinator* for the transaction execution. The coordinator executes the transaction by optimistically reading from different shard leaders (as needed), and buffering the writes. Reads for some key at a given shard leader return the most recent writes to that key, including those that are *certified* (explained below) but not yet replicated, but they will not return uncertified writes.

Certification: After the execution, the coordinator will run a 2PC-based procedure among the shard leaders to confirm that there are no conflicting transactions. The transaction will be assigned a version vector clock. This vector clock represents the serialization order of the transaction. Note that so far no replication has happened, and the transaction is not considered done. Therefore, the transaction "commit" is speculative, with a chance of rollbacks in case of failures. But after certification, writes are speculatively installed and are ready to be read by later transactions. Hence, if rollbacks happen, they can have a *cascading* effect (one rollback triggering another rollback).

Replication: Each shard runs multiple Paxos instances, one for each core (worker thread), to replicate transactions' logs to followers after certification. To maximize throughput, the replication process is completely independent for each shard/core—different shards/cores have zero coordination.

Replay: The followers need to replay from the per-core log to reconstruct the same state as the leader. Because the replication phase skips coordination across shards and cores, the dependency information across shards and cores is missing. Without such information, the replay could lead to inconsistent states. To address this, Mako designs a decentralized and lightweight mechanism, the *vector watermark*, to track dependencies of distributed transactions for safe replay (§4.3).

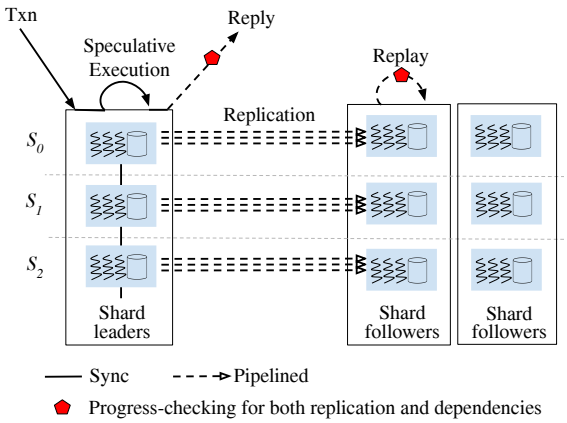


Figure 1: The architecture of Mako. Shard leaders can be in the same datacenter or in different datacenters. Shard followers are in other geographic locations. Results are returned to the client only after replication completes.

4.2 Speculative execution and certification

Mako extends Silo's [95] single-node optimistic concurrency control (OCC) protocol to a distributed variant of OCC. Every shard leader in Mako can receive a *one-shot transaction* (i.e., a transaction that includes all operations, sometimes termed a non-interactive transaction) from a client. Upon receiving that request, the shard leader becomes the transaction coordinator and will execute the transaction on behalf of the client. For reads, the transaction coordinator performs optimistic reads of key-value pairs. It maintains a ReadSet, which includes all the records read with their *versions* (defined below). For writes, the coordinator buffers them into a WriteSet that stores the new state of the record locally without explicitly coordinating with other transactions; these are not installed until they have been validated. Records that have been both read and modified are present in both the ReadSet and WriteSet. In the absence of failures, reads recorded in the ReadSet reflect the latest version of the corresponding keys.

Version vector clock. The version in Mako is a vector clock that consists of n individual logical clocks $(c_0, c_1, ..., c_{n-1})$, where n represents the total number of shards in the deployment. Each element represents a logical clock value generated by a shard. The logical clock on each shard increases monotonically. In our implementation, this is generated by an atomic `fetch_and_add` instruction as it fits our needs. If higher per-shard scalability is desired, one can use the `rdtscp` instruction as done in previous works [51, 66, 85, 92]. Further, Section 6.1 discusses how Mako represents this clock when there are thousands or millions of shards to keep the system scalable.

Transaction certification. After a transaction executes, the system needs to certify that there are no conflicts with concurrent transactions. The certification includes 4 rounds of RPCs among the leaders of involved shards (single-shard transactions perform no RPCs).

1. *Lock*. The coordinator attempts to acquire locks for the records in the WriteSet from the involved shards. Lock requests are sent in parallel. During the lock acquisition, if any lock is already held by another transaction, the lock request fails and the attempting transaction is aborted.

2. *GetClock*. After locking the WriteSet, the coordinator asks each involved shard to increase and return its latest logical clock to create a vector clock. The coordinator then combines this vector clock with all the vector clocks in the ReadSet to generate a vector clock that serves as the transaction's commit version. The rule for combination is choosing the maximum clock value for each shard.

3. *Validate*. The coordinator then contacts the relevant shards to check all the records in the transaction's ReadSet. If there is a conflict on any of the keys (i.e., the latest version has changed or the lock bit has been taken), the coordinator will abort the transaction. This is to ensure that the versions of all read data at the serialization point are the same as the versions seen during the execution phase.

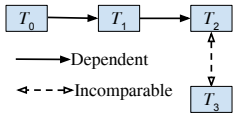
4. *Install*. If all validations succeed, the coordinator sends RPCs to the relevant shard leaders to speculatively install the writes. Old versions are not discarded immediately in case the speculatively installed versions need to be rolled back.

Validate and *Install* are essentially 2PC's prepare and commit phases. We choose not to use 2PC's naming to avoid confusion about the commit point, as the transaction is not fully "committed" yet since it is pending replication.

After the above 4 RPCs are done the certification phase is complete and the transaction's state moves to `CERTIFIED`. So far, all communication has occurred among the shard leaders. If the leaders are co-located, this means that no cross-datacenter communication needs to take place and Mako can use fast networking (DPDK).

Use vector clock to coarsely track dependencies. Mako needs to track read dependencies between transactions that have been certified and speculatively installed to prepare for possible rollbacks. If transaction T_1 reads T_2's write, then T_1 (read-)depends on T_2; if T_2 reads from T_3, then T_1 transitively depends on T_3. If T_1 (transitively) depends on T_N, Mako ensures that T_1's version vector clock is always greater than T_N in a pair-wise manner. *This is the key invariant in Mako that ensures the correctness and the efficiency of rollbacks*. (If T_1 is a read-only transaction, T_1's version could equal T_2's.)

For example, in Figure 2 there are 3 shard leaders S_0–S_2, and each shard stores 3 keys respectively. The dependency relationships are as follows: T_2 depends on T_1, T_1 depends on T_0, and there is a blind write between T_3 and T_2. A possible set of versions for this scenario could be T_0: (1,0,0), T_1: (1,1,0), T_2: (1,2,1), T_3: (0,3,0). In this example, $T_0 \leq T_1 \leq T_2$ in pair-wise comparison, while T_3 and T_2 are incomparable, indicating that they are not read-dependent with each other (e.g., commutative or blind write). Note that T_2 must be pair-wise comparable to T_0 as they are transitively read-dependent.



	Operations	ReadSet	WriteSet	Vector Clock
T_0	W(a);		(1,0,0)	(1,0,0)
T_1	R(a);W(d);	(1,0,0)	(0,1,0)	(1,1,0)
T_2	R(d);W(d);W(g);	(1,1,0)	(0,2,1)	(1,2,1)
T_3	W(e);		(0,3,0)	(0,3,0)

Figure 2: Transitive transactions: version vector clocks reflect all potentially explicit or implicit dependencies. There are 3 shard servers $S_0 - S_2$, and 4 transactions: T_0–T_3.

If two transactions are incomparable, our vector clock still reflects the correct serialization order from the leader (e.g., T_3 has to be replayed after T_2 on the followers of shard S_1).

4.3 Replication with Paxos streams

There are two types of logs in Mako: a *transaction log* and a per-core *replication log* (which we call a *stream* to avoid ambiguity). A transaction log contains the key-value pairs in the WriteSet of the transaction, and its commit version vector clock. In contrast, each entry in a stream corresponds to a batch of transactions (400 in our implementation).

Once a transaction is speculatively installed in the shard leaders, the transaction enters the replication phase. During this phase, each worker thread within each shard maintains a separate stream of the transactions that will be replicated. Mako then uses MultiPaxos [16] for replication (a separate instance for each core), but it could use any leader-based replication protocol that is safe under an asynchronous network.

Note that we use per-core streams rather than a single stream for the entire shard because prior works [85, 90] have shown (and we have confirmed) that the throughput of a single MultiPaxos stream plateaus after $\approx$ 10 worker threads due to expensive thread synchronization overhead. Further, the reason that each stream entry corresponds to a batch of transactions instead of a single transaction is to eliminate the frequent RPC overhead; all other works in the literature do the same. While batching could introduce additional latency, Mako operates at high throughput so it builds batches fast; the geo-latency of replication is by far the dominant contributor to the end-to-end latency experienced by clients.

4.4 Record-Replay on the followers

When a new Paxos stream entry is durable, a follower cannot replay the entry just yet since this might lead to an inconsistent state in the event of failures. Consider two transactions T_1 and T_2 in Figure 3. The system has two shards to store bank checking and saving information separately. Initially, Alice's savings account holds \$100, while Bob has a bank balance of \$0. In T_1, Alice transfers her balance from the savings account to the checking account, and then in T_2, Alice transfers \$100 from her checking account to Bob's checking account. T_1's two operations will be replicated independently by two Paxos streams as they are on different shards. However, in the event that the saving account shard fails, two potential outcomes may arise, both of which can lead to an inconsistent

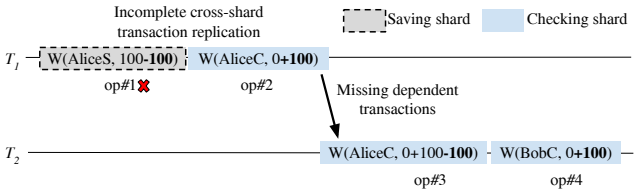


Figure 3: Alice withdraws $100 from her savings account to her checking account and then transfers it to Bob.

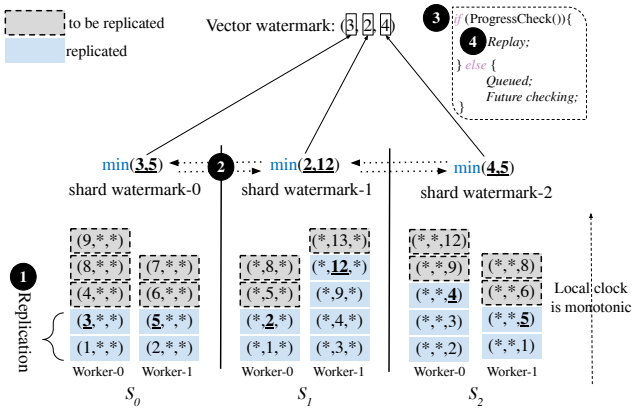


Figure 4: Example replay procedure for shard followers with 2 worker threads per shard. Replay is performed by advancing the vector watermark. Mako only tracks the clock relevant to each shard (we mark the irrelevant entries with *).

state. The first outcome is *incomplete cross-shard transaction replication*. For instance, op#2 in T_1 is successfully replicated, while op#1 is lost after a failure. It is unsafe to only replay op#2 on one shard follower. The second outcome involves *missing dependent transactions*. Although all operations in T_2 are replicated, T_1's may not be (i.e., op#1 is lost). Mako cannot replay T_2 either, as otherwise the system will be in a state where Alice never sends money, but Bob receives $100.

To address this issue, Mako borrows the idea of Aurora [99], Rolis [85], and Query-Fresh [99] in which system performance is optimized by pipelining replication and utilizing one monotonically increasing number to track the durability of concurrent transactions within one shard or one database instance. However, this approach will not work out of the box in multi-shard, geo-replicated systems since the monotonically increasing number would become a centralized bottleneck. To correct this, Mako introduces a lightweight, decentralized *vector watermark* scheme to enable safe replay.

A vector watermark is an array of shard watermarks (w_0, w_1, ..., w_{n-1}) where n equals the number of shards. Each shard watermark w_i is calculated by shard-i independently. On shard-i, Mako maintains an invariant that each Paxos stream has a local and monotonically increasing clock value that represents the shard clock of the most recently replicated transaction. Shard-i chooses the minimum of these clock values from all worker threads. Figure 4 shows an example.

❶ Each watermark w_i represents the current replication progress of shard-i in the system. It is monotonically increasing. ❷ All shards will periodically gossip with each other to exchange their latest vector watermark. The gossip is performed in the background and does not block transactions' execution. ❸ Once *progress-checking*, a safety-checking mechanism, succeeds at the followers, ❹ the transactions are sent to the replay threads for concurrent replay. We discuss progress checking and concurrent replay in detail later.

For a coordinator to acknowledge the transaction's result to the client, Mako waits for the system's vector watermark to advance beyond the transaction's version. This ensures that the transaction and all transactions it depends on have been durably recorded on all relevant shards, providing a consistent and reliable commit acknowledgment to the client. More formally, if a transaction with version vector clock $(c_0, c_1, ..., c_{n-1})$ has $c_i \leq w_i, \forall i \in [0, n-1]$, its transaction log must be already replicated in the Paxos stream of shard-i. With this guarantee, a transaction can transition to `COMMITTED` and return back to the client.

Progress-checking phase. In this phase, Mako actively verifies whether a transaction's version falls below the latest vector watermark. If it does, the transaction proceeds to the replay phase, indicating that all transactions it depends on, from other cores or shards, have also been replicated to a majority of replicas. At this stage, Mako can assert that the transaction is `COMMITTED`. On the other hand, if the version does not fall below the vector watermark, the transaction is unsafe to replay since some of its dependencies may be missing, or some transactions might only be partially replicated. In such cases, the transaction is queued for future checking.

Replay phase. We employ Thomas's write rule [90, 93] (e.g., last writer wins) to replay replicated writes on each shard follower concurrently without any additional coordination. Each replicated write corresponds to one transaction. If the transaction tries to write a data item that has already been written by another transaction with a newer *shard clock*, the write operation is ignored.

Optimization: summary vector clocks. Since we batch transactions to avoid RPC overhead, we also find it helpful to do watermark updates and comparisons at the granularity of batches instead of individual transactions. We do this by assigning each batch a *summary vector clock*. Mako computes this summary vector clock by using the highest c_i for all $i \in [0, n-1]$ in all vector clocks in the logs of the stream entry. For example, if we have a batch that is made up of the 4 transactions in Figure 2, the batch's summary vector clock would be $(1, 3, 1)$. This vector clock is used when the followers replay operations.

5 Handling failures in speculative 2PC

The speculative transaction execution and pipelined replication strategy discussed so far assumes no failures. This section discusses how Mako deals with the cases where speculation fails due to server failures. We only discuss failures of shard leaders as failures of followers do not require special handling other than normal recovery.

5.1 Why is this hard?

A conceptually simple strawman solution is to pause the entire system, sync all replicated transaction logs, track which speculatively executed transactions are not fully replicated, rollback these transactions and any transactions that depend on them, and then resume execution. While straightforward, this approach is not ideal in practice as a single failed server can halt the entire system. Furthermore, in multi-shard systems used today such failures do not impact healthy shards. Mako preserves this arrangement by ensuring that unaffected shards can continue processing transactions.

Our original intent was to find and then adapt an existing technique in order to avoid freezing the entire system in the event of speculation failures. We were confident that given the long history of work in speculative execution, such a solution must exist. To our surprise, we found no prior solution that could be applied to Mako, and realized that Mako faces unique challenges. Below is a summary of prior solutions and why they do not apply to Mako.

- *Single-shard speculation*. Many speculative execution and replication works only consider a single shard [33, 81, 85]. In speculation failures, the system freezes to recover. There is no need for a better solution because there is only one shard in the system so the system would have to freeze anyway when the shard fails.
- *Re-execution*. Two recent designs, Morty [12] and Hackwrench [24], use speculative execution on transactions and re-execute the transaction if the speculation fails. An important reason why they can do this is that their speculation failure is not caused by server failures but by conflicting access in transactions. That is, even though the speculation fails, the transaction requests/logs are still present to support the re-execution. But in Mako, the servers can fail before the replication completes, so transactions could be lost; Mako cannot re-execute transactions in the way that Morty and Hackwrench do.
- *Fine-grained dependency tracking*. By recording the fine-grained dependencies (e.g., T_1 reads T_2), the system can accurately abort transactions in a cascading manner. However, recording such dependencies has a high overhead and thus is only used in distributed systems [68, 105] where the baseline throughput is relatively low (i.e., thousands of transactions per second) so there is room to tolerate the overhead. In contrast, multicore transactional systems have much higher throughput (millions of transactions per second), so introducing fine-grained dependency tracking can easily kill more than half of the throughput, as is discussed by prior works [23, 28, 34, 100]. Hence, if we want Mako to be fast, it cannot take this approach.
- *Group-commit*. Designs that execute transactions speculatively and commit all of them as a group [54, 64, 101] track replication progress with a single timestamp. But if a shard leader fails, the entire system blocks until the failure is detected and fixed (e.g., several seconds). Mako cannot use this approach because we want healthy shards unaffected by the failure to continue executing transactions.

Our literature survey leads us to conclude that no existing solution can meet Mako's goals. A deeper reason for the absence of techniques in this domain is that the key building block in most distributed transaction protocols, 2PC, is not fault-tolerant. If a shard in 2PC loses its state, the healthy shards will also get stuck [87]. The usual approach to address this problem is to replicate every decision in 2PC [20] or rely on fault-tolerant disaggregated storage [35] so that we can assume that every shard is always alive. But since Mako chooses to invert the layering between 2PC and replication, we must invent a new failure recovery mechanism to support recovering 2PC with incomplete past decisions.

5.2 Mako's solution

As Mako does not replicate 2PC's decision before it finishes, Mako cannot recover decisions at the level of individual transactions when there are failures. But such fine-grained failure handling is not necessary anyway. Instead, Mako groups transactions into *epochs*. When failures happen, Mako advances the epoch and makes a collective decision about which transactions in the previous epoch must roll back.

Mako has a configuration manager (CM) that manages the epochs (which is also sometimes called a view manager in other works). CM itself is replicated so it is considered always alive. CM maintains heartbeats with every shard. When a shard leader fails, CM will detect it and trigger a leader election to elect a new leader. CM also advances the epoch number and broadcasts the epoch increment to all shards. The Paxos streams on all shards also use the epoch number to group log entries, which is a classic approach in consensus systems [16, 72, 73]; we are not introducing algorithmic changes in reaching consensus.

The newly elected leader will first try to retrieve all replicated log entries in the previous epoch from its peers, recommit them if needed, and commit any remaining slots with no-ops if the entries are not recoverable (any entries after the no-op will be ignored). We call this *closing* the previous epoch, because after this step no more log entries can be added to the previous epoch. When an epoch is closed on a shard, its *finalized* shard watermark can be computed by choosing the minimum shard clock of all its streams.

Healthy shards also advance their epochs on receiving the broadcast from the CM. A major difference between a healthy

shard and a recovered shard is that a healthy shard usually does not lose any transactions in its log, so its rollback is expected to be less aggressive. To reflect this, the healthy shard has an extra step to close the old epoch. The shard first tries to finish speculative execution and certification of all transactions in the old epoch, then it replicates a special ending entry into all its Paxos streams. The ending entry has a INF shard clock value, indicating that: (1) this is the maximum clock; (2) there are no more transactions of this epoch on this shard; and (3) all previous transactions have successfully been replicated without lost dependencies. When the replication of INF entries finishes, the leader considers the old epoch closed, and declares its finalized shard watermark is INF. Note that INF is also the minimum shard clock of all of its streams, so the rule for computing the finalized shard watermark is the same on recovered shards and healthy shards.

If the shard has a hanging transaction in the old epoch that cannot finish before a timeout (due to network delays or waiting on the failed shards), it will terminate the hanging transaction, but not replicate the INF entry in the Paxos stream, and then computes the watermark normally.

After a shard computes its finalized watermark, it broadcasts the watermark to all other shards. Eventually, all shards exchange their finalized watermarks to form the *finalized vector watermark* (FVW), which represents a consistent and maximum global cut across all shards for the old epoch. Given an epoch, all possible vector watermarks are smaller than the epoch's FVW in a pair-wise manner. Computing the FVW is a deterministic and reentrant process, meaning it can be safely restarted and re-executed in case of failures, always yielding the same result. If a shard fails during the steps of closing an old epoch, the CM will initiate a further configuration change with a higher epoch. The new leader will always compute the same FVW for the old epoch as they were computed before.

After the FVW for the old epoch is established, any transactions that are not below the FVW are rolled back on shard leaders and forever abandoned by the system. One might wonder why we cannot re-apply the transactions using the logs from healthy shards. This is because although healthy shards may have a transaction's WriteSet, these writes can depend on lost transactions, so re-applying them would lead to anomalies (§4.4). Note that the number of transactions above the FVW in the old epoch does not grow once the FVW is established, which means the rollbacks are bounded; there are no unbounded and expensive cascading aborts into new epochs.

During this failure recovery, transaction execution on healthy shards should not be blocked if transactions do not involve the failed shard. To reflect this, a healthy shard does not need to wait for the old epoch to close, and the finalized vector watermark to be computed, before processing new-epoch transactions. It does need to wait until all old-epoch transactions are certified or aborted (but not replicated), so that no old-epoch transactions read new-epoch transactions. This switch is very quick. To prevent new-epoch transactions

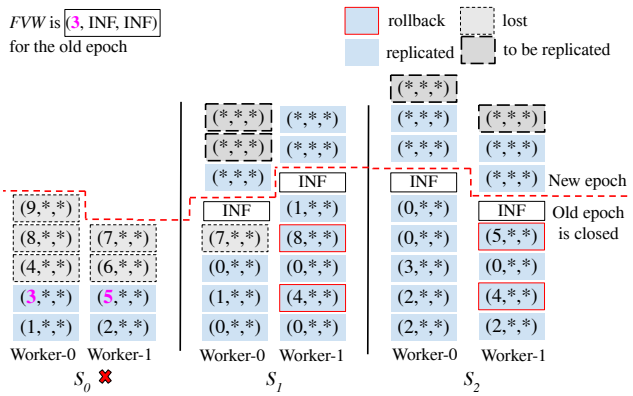


Figure 5: Single-shard failure: S_0 fails; S_1 and S_2 are healthy (irrelevant information is marked with asterisks).

from being affected by the old-epoch rollbacks, we add a rule that before the FVW is computed and the rollback is done, new-epoch transactions cannot read from transactions that are certified but are uncertain if they will be below FVW.

Figure 5 shows an example. There are 3 shards with shard S_0 failing and triggering a rollback. The computed FVW is (3, INF, INF). All healthy shards (S_1 and S_2) can eventually replicate their transactions in the old epoch. For any transactions on the healthy shards that are not below the FVW, such as (4,*,*), they are rolled back because they could transitively or directly depend on one of the lost or incomplete transactions.

Additional details. After all shards have advanced to the new epoch and all old-epoch transactions have finished, the healthy shards can clean up leftover work by the failed shard, mainly its pre-owned locks. Further, if a healthy shard launches a transaction that accesses a shard that is in the process of recovering from a failure, the healthy shard may be blocked because Mako has a synchronous threading model. To prevent this, Mako puts this transaction into a wait queue based on timeouts until the failed shard recovers.

Correctness. We give a proof of the correctness of Mako's design and failure recovery in Appendix A.

6 Practical considerations

In this section we discuss critical extensions to Mako that are needed for Mako to be useful in practice.

6.1 Scalability with more shards

Recall that Mako creates shards for 2 reasons: to achieve higher throughput and to support increasingly large datasets that cannot fit in one machine. As we show in our evaluation (§7), Mako achieves very high throughput with just a handful of shards and hence there is no need to use many shards to get high throughput. On the other hand, Mako absolutely needs to scale to support larger datasets.

But adding more shards to Mako brings overheads that can prove problematic. In particular, there are two components of Mako that scale with the number of shards: (1) the vector

watermark introduced in Section 4.4 that tells followers when it is safe to replay operations; and (2) the version vector clocks that track the version of objects across transactions. *The vector watermark is not a scalability bottleneck*, because each shard maintains a single vector watermark (the number of entries is linear in the total number of shards), and each entry in it is a 32-bit integer. Even if we had 10,000 shards, each vector watermark would be only 40KB.

On the other hand, there is a version clock for every object in every transaction. *Version vector clocks are a scalability bottleneck*. To address this issue, when the number of shards gets large, Mako compresses the vector clocks to a small size similarly to prior works [57, 102, 106, 110]. We provide a complete pseudocode for the algorithm in the Appendix D, and evaluate the scalability of vector clocks in Section 7.7. The high-level idea is as follows.

When installing a new version vector clock, multiple shards' clocks can be merged into one entry by choosing the maximum of these shards' clocks (the shards' logical clock is also updated to the maximum value on installing the version). This compression is lossy, but it still preserves correctness because it maintains the key invariant in Mako: if transaction T_2 (transitively) depends on T_1, T_2 must have a greater vector clock. The merging strategy is flexible: Mako can merge any group of clocks into one. For simplicity, we implemented a $K:M$ strategy that groups K shards into M-sized vector clocks.

Note that compression has its downsides: it degrades performance when a shard fails or slows down. Consider the extreme case when the vector clock is compressed to a single timestamp. In this case, when a shard wants to replay a transaction, it needs to wait for the watermark of every other shard to surpass the transaction's timestamp. If any shard is laggy or faulty, the whole system is affected. In contrast, in the uncompressed case, replaying only needs to wait for relevant shards, not all shards. Mako uses a full-sized vector clock as its default; when there are many shards it sets the vector clock to a constant number (e.g., 20) and merges clocks.

6.2 Shard management

Similarly to most works in this space, we use static sharding in our prototype implementation to focus on our primary research question. However, shard management, such as resharding, is an important topic in practice [20, 41]. Prior works [1, 6, 48] have shown that thoroughly addressing shard management requires an entire research project.

We leave the complete design and implementation of a dynamic sharding solution for Mako to future work. However, we argue that this problem is not fundamentally more difficult in Mako than in prior systems. In particular, in Mako, there is no need for cross-epoch vector clock comparisons and cross-epoch vector watermark computations, which eliminates concerns about mismatches between vector clocks across epochs. Leveraging this insight, Mako is able to handle shard management in a way analogous to failure recovery. Once a FVW is computed (§5.2), the entire database can be treated as a read-only snapshot, serving as the initial state with all vector clocks conceptually reset to zero as vector clocks from the higher epoch always have a higher priority. We provide more details in Appendix B.

6.3 Quick failure recovery

Learners. To speed up failure recovery, Mako co-locates a shard learner in the same datacenter as the shard leader. A shard learner is essentially the same as a shard follower, except that it does not vote on consensus. This allows Mako to recover from the failure of a leader more quickly than having to make one of the followers the leader (which would cause shard leaders to no longer be co-located) or provision a new leader in the existing datacenter which would take some time for this new machine to catch up from scratch.

Datacenter failures. A special case in deployment is when all shard leaders are co-located in the same datacenter, and the entire leader datacenter fails. In such a case, the failure recovery process can be expedited by skipping all rollbacks on healthy shard leaders since there are none.

Stragglers. A benefit of using a vector clock is that it is resilient to stragglers. A straggler shard in Mako does not block the system entirely. Note that the all-to-all communication in Mako for vector clocks is non-blocking. If a shard is very slow, transactions that do not involve such straggler shard are not affected. Recall that the watermark is a vector: the straggler's entry in the vector would stay the same until the straggler makes progress, but the other entries in the vector would be incremented. So transactions that do not touch the straggler can continue to be replayed at followers and can return to the client. Only when cross-shard transactions touch the straggler, would they incur any delay.

7 Evaluation

The evaluation aims to answer the following questions:

- What is the base performance of Mako and does it scale?
- How does Mako compare to the state-of-the-art geo-replicated transactional systems?
- How fast can Mako recover from failures?

7.1 Experimental setup

We implemented Mako in C++. Mako is based on a few existing works, including Silo [95] for each server's database engine, eRPC [45] for accelerated networks, and the Janus framework [69] for replication. Mako adds ∼10K new lines of code. Mako is open sourced at `https://github.com/stonysystems/mako`.

Testbed. Experiments were conducted on Azure. Each VM has 32 Intel vCPU cores, 128GB of RAM, and Mellanox 4 Lx accelerated networking (16 Gbps). The latency within a datacenter is mostly between 20 and 30 μs, with occasional

peaks over 100 μs.[3] To simulate WAN, we make 3 groups of servers and inject a 50 ms RTT between groups.[4]

Baselines. We evaluate nine baselines: (1) five systems that support scalable sharding and geo-replication, including Calvin [94], D2PC [113], TAPIR [109], a Spanner-like 2PC over Paxos, and Janus [69]; and (2) four systems that lack full sharding or geo-replication features but are optimized for multi-core settings, including Silo [95], Meerkat [90], Rolis [85], and DrTM+R's Optimistic Replication (OR) [104]. For Calvin, D2PC, TAPIR, Janus, Silo, Meerkat, and Rolis, we use their existing implementations with modifications required to get them to run in our testbed (e.g., we had to port Meerkat's networking drivers to work with the network cards in Azure). We implement our own version of OR because the original implementation does not support geo-replication. We add this support by combining OCC with DrTM+R's optimistic replication protocol (we call it OCC+OR). We also implement a Spanner-like 2PC over Paxos since Spanner is not open source (we call it 2PC).

Sharding in Mako and baselines. The database is split across anywhere from 1 to 10 servers. Given that many of the baselines were not designed to run on many-core machines, we need to be careful with how we shard the data as otherwise we could be very unfair to them. We do the following.

- **Mako, OCC+OR, 2PC**: 1 server is 1 multi-core shard with 24 worker threads.
- **Janus, D2PC, TAPIR**: 1 server is 24 shards, each with 1 worker thread.
- **Calvin**: 1 server runs 3 shards (called "partitions" in Calvin), and each shard has 8 worker threads.

The goal of the above arrangement is to ensure that, to the best of our ability, we are giving the same amount of hardware resources to all of the different systems. There is however, another issue that arises. When we run many of these baselines with many servers, they end up having a lot of shards. For example, if we run Janus, D2PC, or TAPIR with 10 servers, that is equivalent to running them with 240 shards. The research prototypes of these systems were never tested with such high number of shards, so at this scale all of them experience issues (e.g., can't establish socket connections, abort rates are too high, most RPCs time out) when performing cross-shard transactions (we have confirmed this with several of the authors of these systems). We therefore make the decision to be **generous** to these baselines and disable all cross-shard transactions for Janus, D2PC, TAPIR, and Calvin. In other words, they only perform single-shard transactions, whereas Mako, OCC+OR, and 2PC will perform cross-shard transactions.

To ensure fault-tolerance, we also replicate each server so we have one leader, one learner (§6.3), and two followers (this means in total we have up to 40 servers). Unless otherwise noted, all shard leaders are deployed in the same datacenter, and all clients are co-located with the shard leaders.

Benchmarks. We evaluate two benchmarks:

- **TPC-C**. This benchmark simulates an e-commerce site. TPC-C consists of a mix of five concurrent transactions that represent different types of activities, including new orders (NEW), payment processing (PAY), order status checking (ORDER), delivery scheduling (DLY), and stock level checking (STOCK). TPC-C scales by sharding a database into multiple warehouses spread across multiple cores and shards. Transactions in TPC-C are configurable to span multiple warehouses to evaluate the performance and scalability of distributed database systems. This type of transaction is more complex and resource-intensive than transactions that access data within a single warehouse, as it requires coordination and communication between multiple cores and servers. We run the standard mix on the default configuration unless otherwise mentioned.
- **Microbenchmark of small-sized transactions**. Because Mako introduces overheads at the transaction level, its performance is penalized more with small-sized transactions. We demonstrate cases like this using a microbenchmark with two transaction types: Read-Only (READ) and Read-Modify-Write (RMW). Each READ transaction has 4 read operations and each RMW transaction has 4 read-modify-write operations. The total data space for the microbenchmark is 1 million keys for *each* shard. We conduct 50% of RMW transactions and 50% of READ transactions. Each operation randomly selects a key in a given shard. There is a 95% chance of selecting from the shard where the coordinator is running, and a 5% chance of the coordinator performing a cross-shard key access.

7.2 Throughput and scalability

This part evaluates the performance and scalability of Mako.

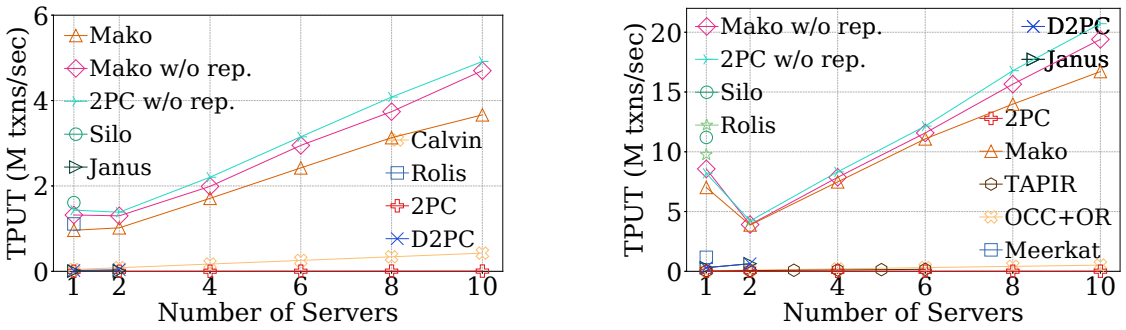
TPC-C Benchmark. Of the 10 systems that we are evaluating, only 7 systems readily support TPC-C. These include: Mako, 2PC, Janus, D2PC, Calvin, Silo, and Rolis. TAPIR and Meerkat do not support TPC-C in their existing code bases, and we did not implement TPC-C for OCC+OR.

We run TPC-C on the 7 systems that support it with an increasing number of shards. The number of warehouses is set to be equal to the number of worker threads in the system.

Mako is able to scale well in Figure 6a. Mako's throughput at 10 shards (10 servers) is 3.66M transactions per second (TPS), which outperforms Calvin by 8.6×. As a side note, Mako also has good single-shard performance reaching 0.96M TPS, which outperforms Calvin by 22.5×. The major reason for the slowness in Calvin is that it uses a central se-

[3] Note that this is higher than the usual single-digit microseconds latency one would observe in a local cluster with the same high-end networks. After consulting with Azure administrators as well as authors of other works using Azure [27], we learned that the higher latency on Azure is the norm and we assume it is related to the virtualization infrastructure.

[4] We inject latency instead of deploying in multiple datacenters since we were limited by Azure quotas. We tested small-sized deployment in multiple datacenters and the results are consistent with the injected-latency test.



(a) TPC-C Workload **(b) Microbenchmark workload**

Figure 6: Throughput and scalability of Mako and the baselines. In Mako, the number of shards is equal to the number of servers (x-axis). In other systems, the number of shards is larger than the number of servers (see Section 7.1).

quencer to pre-determine the order of a batch of transactions, which are then sent to all replicas to execute deterministically through Zookeeper.

There is a noticeable per-shard throughput drop when increasing the number of shards in Mako from 1 to 2. The reason for this decline is that 1% of the items accessed in the `NewOrder` transaction (10 items accessed on average) trigger a cross-warehouse access, which results in 5.11% cross-shard transaction coordination. This overhead is considerably more expensive compared to the local in-memory operations.

The performance of Mako scales almost linearly after 2 servers. As the number of servers increases, Mako's throughput per shard slightly decreases with increasing shards from 2 to 10 shards, because the ratio of cross-shard transactions increases. For example, `NewOrder` goes from ∼5% to ∼9% (the theoretical bound is 10%). While 2PC scales well up to 10 servers, its throughput is lower than Mako because geo-replication is on the critical path of concurrency control. This same effect of having geo-replication in the critical path affects D2PC and Janus, which can only reach 38.5K TPS and 10.4K TPS with 2 servers, respectively.

Ablation testing. To understand the contribution of different components in Mako, we disable various parts of Mako and some of the baselines.

First, we disable all replication for Mako and 2PC. We observe that both Mako and 2PC without replication perform much better than their respective variants with replication (as is expected). For example, the overhead of replication in Mako is around 23%. 2PC, without the geo-latency to slow down its transaction processing, gets a massive performance boost and is even better than Mako because it has fewer RTTs for cross-shard transactions.

Second, we measure Silo, which is the system on top of which we built Mako, to investigate the cost of adding sharding and replication to a single-machine database. Mako preserves 68.4% of the throughput of Silo on a single machine.

Third, we evaluate Rolis, which extends Silo to support geo-replication efficiently. Rolis achieves higher throughput than Mako with 1 shard since it does not have multi-version overhead. However, Rolis can only support one shard so if the database gets too big or if the system requires more throughput, there is nothing that Rolis can do besides using bigger machines. Adding support for sharding is precisely Mako's contribution over Rolis.

Microbenchmark: We further study the performance and scalability of Mako on the microbenchmark with a fixed 5% cross-shard key access, as shown in Figure 6b. In this experiment, we also include the results for OCC+OR, Meerkat, and TAPIR. In this simple but high throughput scenario, Mako can still scale well beyond 2 shards. The throughput of Mako at 10 shards is up to 16.7M TPS. Both Mako and OCC+OR can scale well with more shards, but Mako is able to achieve 32.2× higher throughput than OCC+OR at 10 shards (10 servers). OCC+OR can only achieve up to 0.52M TPS because transactions in OCC+OR always optimistically read and abort if the replicas are not ready during the commit phase, which is often the case under geo-replication. As in the TPC-C benchmark, Rolis achieves 39% higher throughput than Mako when using a single shard.

Meerkat does not support sharding, so we test its single-shard performance with 24 worker threads. All servers are connected via DPDK. Meerkat achieves 1.17M TPS, a lot lower than Mako at 1 shard. In fairness, Meerkat supports interactive transactions which are more costly to handle than the "one shot" (or non-interactive) transaction model used by Mako and the other baselines. So this comparison is not apples-to-apples and should be treated qualitatively (Mako is very competitive) rather than quantitatively.

We test Janus, D2PC, and TAPIR up to 2, 2 and 6 servers, respectively. All systems scale well, with Janus achieving 640K TPS, D2PC attaining 137K TPS, and TAPIR reaching 168K TPS. We could not test beyond these configurations because the prototype implementations of these systems could not support more shards. Nevertheless, assuming they continue to scale in the same fashion, Mako still achieves orders

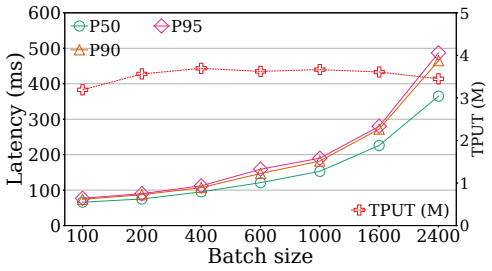


Figure 7: Mako's latency and throughput on TPC-C with varying batch size.

of magnitude better performance by decoupling transaction coordination and replication.

7.3 Latency

Mako's latency has three dominant sources: the RTT between datacenters, the batching window to gather transactions and put them into a stream entry, and the advance time of the vector watermark to go beyond the vector clock. We measure Mako's latency with 10 shards on TPC-C. The results are shown in Figure 7. Median latency with batch size of 600 is 121 ms, of which ∼50 ms are for RTT between datacenters, 13 ms for batching, and the rest to advance the vector watermark. In contrast, 2PC (not shown in the figure) has a much higher latency (∼ 10×) at high throughput because of aborts.

As shown in the results, batch size has an obvious impact on latency but the impact on throughput is minimal once the batch size is large enough (a few hundred). Compared to a batch size of 100, the throughput of Mako with a batch size of 400 increases by 15.8%. This throughput remain roughly constant up to batches of 1600. Based on this, we set Mako's batch size to 400 by default, and used this in all experiments.

We also evaluate the latency of Mako, Janus, and Calvin under a light workload (microbenchmark) using a single replicated shard. This helps us understand the communication overhead of the various systems. Figure 8 gives the results.

The median latency of Mako is 60 ms, comprising approximately 50 ms for one WAN RTT, 3.5 ms for batching, and 6.5 ms for waiting on the watermark advancement. In comparison, Janus achieves a best-case latency of around 50 ms as expected (one WAN RTT), while Calvin exhibits a latency of 166 ms using ZooKeeper-based replication, which matches the number reported in the Calvin paper. All tail latencies are not significantly affected under the light workload.

7.4 Impact of cross-shard transactions

In this experiment, we evaluate Mako on the microbenchmark using 10 shards while varying the ratio of cross-shard transactions to measure the impact of distributed transactions. As depicted in Figure 9, in the absence of cross-shard transactions, Mako achieves a peak throughput of 60.3M TPS, which then declines. When all transactions are cross-shard,

Percentile	Mako	Janus	Calvin
10%	57 ms	50.3 ms	146 ms
50%	60 ms	50.5 ms	166 ms
90%	64 ms	50.7 ms	202 ms
95%	65 ms	50.8 ms	206 ms
99%	66 ms	51.3 ms	212 ms

Figure 8: Latency measurements for Mako, Janus, and Calvin on the microbenchmark under a light workload.

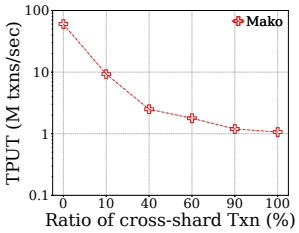


Figure 9: Varying ratio of cross-shard transactions on the microbechmark.

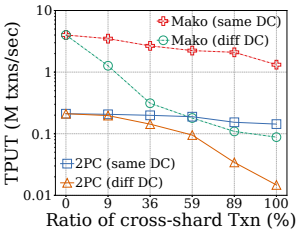


Figure 10: Effect of leader placement on Mako's throughput.

the throughput drops by 98.2% to 1.1M TPS primarily due to the significant time consumed by RPCs.

7.5 Leader placement

We distribute leaders evenly into two datacenters to evaluate how the conflict windows for cross-shard transactions is prolonged and how this affects performance.

We use the `NewOrder` transaction in TPC-C as the benchmark. The ratio of cross-shard transactions is controlled by varying the cross-warehouse access. Since the `NewOrder` transaction accesses an average of 10 items, a 1% cross-warehouse access results in approximately 9.04% distributed transactions, while a 50% cross-warehouse access leads to more than 99% distributed transactions.

In Figure 10, we increase the ratio of cross-shard transactions while fixing the number of clients (24K clients) for Mako and 2PC to test intra-/inter-datacenter transactions. As expected, if there are no cross-shard transactions, the performance of Mako and 2PC are not impacted, reaching 4.0M and 52K TPS respectively. As the cross-shard transaction ratio increases, the throughput of both systems drops significantly. However, even with leaders in different datacenters Mako achieves much higher throughput than 2PC.

7.6 Effect of concurrency on throughput

In the introduction, we remarked that a common way in systems to increase throughput is to process more client requests at the same time. We also stated that this is not actually the case in transactional systems. To gather support for this claim, we follow the experimental setup described in Section 7.5 and evaluate the performance of 2PC on `NewOrder` transactions when the leaders are placed in different datacenters. We in-

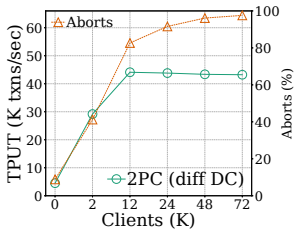


Figure 11: 2PC throughput with varying concurrency.

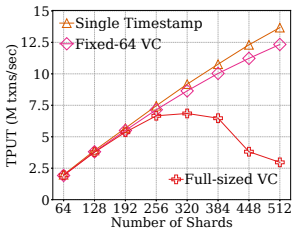


Figure 12: Scalability of vector clock

crease the concurrency (number of clients) while maintaining a fixed ratio of cross-shard transactions (∼9%) in Mako.

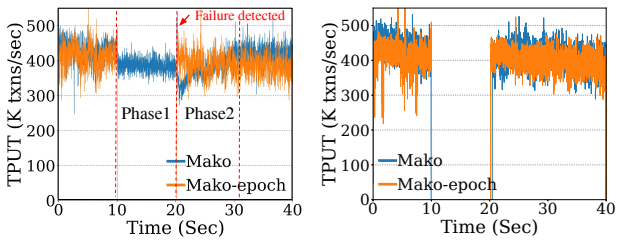
As shown in Figure 11, the throughput of 2PC plateaus, starting with 12K clients. This result shows that simply increasing concurrency for a transactional system does not necessarily lead to higher throughput; instead, it may actually result in lower throughput due to an increased number of aborts. Beyond 12K clients, the throughput remains flat, and the transaction abort rate escalates to an astonishingly high level of 98%. As the contention footprint increases, concurrent transactions are more likely to be aborted due to prolonged conflict windows (at least 1 WAN RTT).

7.7 Scalability of vector clocks

As we discuss in Section 6.1, vector clocks can become a scalability bottleneck when Mako has many shards. To explore this effect and also the improvement of our proposed optimizations, we perform an experiment where we run Mako with hundreds of shards. One caveat is that we do not have thousands of machines available, so our ability to experiment with hundreds of shards (plus their replicas) is limited. We do the following: we limit each shard to use only one CPU and run the microbenchmark. The expectation here is that the throughput numbers will be lower than a standard deployment of Mako, but it should help us appreciate the overhead from the vector clock. Each transaction consists of four RMW operations and 5% of accesses span a different shard.

We test 3 strategies: the full-sized vector clock (VC), a VC that is compressed to a single timestamp, and a VC that is compressed to a VC with exactly 64 entries. The results are shown in Figure 12.

The full-sized VC approach can scale to 320 shards before its performance starts to degrade. This shows the full-sized VC is enough to support many database deployments since Mako is a very high-throughput database, even with a few shards. Of course, if the datasets are massive and the database requires many shards to fit all that data in memory this does mean that the full VC could become a scalability bottleneck. Both the timestamp approach and Fixed-64 VC achieve near linear scalability because the VC-related overhead is independent of the number of shards.



(a) Healthy shard S_1 **(b) Failed shard S_0**

Figure 13: Single-shard failure: TPC-C.

7.8 Single-shard failure recovery

In the case of a single-shard failure, Mako minimizes the impact on uninvolved shards §5.1), which is a key distinction between Mako and systems that perform group or epoch-based commits such as Primo [54] and COCO [64]. To show this difference, we implement a variant of Mako that we call Mako-epoch, which implements the failure recovery strategy of epoch-based commit protocols. Our experiment simulates a shard failure by shutting down shard leader S_0, triggering a learner takeover. Figure 13 reports the throughput of a healthy and failing shard during the failover.

As outlined in Section 5.1, Mako in Figure 13a exhibits 2 phases during failure recovery. Phase 1: the healthy shard remains unaware of S_0's failure and continues sending cross-shard transactions to it. These transactions trigger eRPC timeouts (usually very short, e.g., 5ms) and are queued. After a heartbeat timeout, the healthy shard is aware of S_0's failure. Phase 2: FVW has been computed, and the healthy shard replays transactions from the queue, causing the throughput to gradually climb back to the pre-failure level. Mako in Figure 13b shows the throughput changes in the failed shard. The throughput first drops to 0 for 10 seconds because of the heartbeat timeout, and recovers to a pre-failure level once the new leader takes over.

In Mako-epoch, the behavior for a failed shard is similar to Mako's. However, the behavior for a healthy shard is very different. As shown in Figure 13a, no transactions execute successfully in Mako-epoch during Phase 1 because healthy shards remain unaware of the failed shard(s), and all transactions that execute in the old epoch eventually need to roll back upon failure detection. This is because, in epoch-based protocols like Mako-epoch, an epoch is the commit unit.

7.9 Factor analysis

In Section 7.2 we had an initial ablation study. Here we expand on it to understand exactly how each component contributes to Mako. This test is done with 10 shards.

	Silo	+Multi Version	+Distr. Trans.	+Rep.	+Replay (Mako)
TPUT (M)	1.66	1.48	0.47	0.36	0.36

The starting point of Mako is Silo, which achieves 1.66M

TPS on a single machine without any sharding or replication. The versioned values technique for rolling back results introduces a 11.1% overhead because each data item must now store a list of all its versions, which takes up additional memory operations. In addition, cross-shard coordination for distributed transactions accounts for a significant loss of 68.1%. Mako then needs to use CPU resources to serialize transactions into batched logs and use Paxos streams for replication, which causes a further 22.5% throughput loss. Replay in contrast has no impact on throughput because Mako replays transactions on shard followers asynchronously.

8 Further discussion

Read-only transactions. Our current implementation does not let read-only transactions execute directly at the followers—which is a common optimization. But Mako is compatible with this optimization by employing the high-level approach proposed in Silo [95]. The idea is to keep a dedicated checkpointing thread that periodically snapshots the databas so that read-only transactions can read from this snapshot. To do this in Mako, we would maintain two special versions of each key: one reflecting the latest value and the other storing the most recent value up to the last all-agreed vector watermark for read-only transactions.

High contention workloads. High contention workloads lead to high abort rates in all systems, including Mako. That said, Mako's pipelined replication and smaller conflict window (due to decoupled replication) lead to fewer aborts compared to other systems where replication is in the critical path. While high contention workloads would increase the transaction abort rate, they do not increase stress on Mako, as fewer transaction logs are generated.

9 Related work

This section reviews works in the literature on transactional, geo-replicated, multi-core systems from the various angles.

Distributed systems within the datacenter. Many works improve the performance of distributed transactions within a datacenter [4, 9, 10, 17, 25, 46, 67, 74, 84, 90, 103, 107, 110, 112]. These systems use advanced hardware, such as HTM, RDMA, and DPDK, to deliver good performance, strong consistency, and fault tolerance. A key feature shared by these systems is that replication happens side-by-side with transaction execution and/or commit, making transaction backup part of the critical path. In contrast, Mako is specifically optimized for geo-replicated databases and can therefore withstand full datacenter failures (whereas these works cannot), while still achieving high performance through speculative execution.

Geo-replicated distributed systems . Recent state-of-the-art distributed systems [7, 31, 40, 43, 44, 55, 56, 62, 70, 83, 97, 111, 113–115] pursue fast geo-replication. For example, Ocean Vista [30], TAPIR [109], and Janus [69] only need one WAN RTT if no conflicts are present by combining concurrency control and replication. However, they integrate replication into transaction execution, significantly impacting performance due to high WAN latency.

Speculative replication. Speculative replication [47, 54, 64, 81, 85] is a common optimization technique used in geo-replicated transactional systems to enhance throughput and reduce latency. Besides the discussion in Section 5.1, many systems [3, 32, 58, 61, 77, 88, 108] resort to weaker consistency semantics (e.g., causal consistency) that permit an asynchronous replication strategy in geo-replicated systems. Epoch-based commit systems, such as COCO [64] and Primo [54] speculatively execute transactions within each epoch while asynchronously replicating writes only at epoch boundaries to enhance performance. Compared to these systems, Mako's major advantage is that it does not have to stop healthy shards during a failure recovery as we show in Section 7.8.

Deterministic algorithms. Deterministic databases [29, 37, 39, 80, 94] can execute transactions deterministically across multiple replicas with either asynchronous or synchronous replication. In systems like Calvin [94] and C5 [37], transactions undergo a sequencing phase before execution. This sequencing layer establishes the ordering of submitted transactions and replicates them asynchronously in batches. In contrast, Mako adopts a different approach that eliminates the need for a predetermined order during execution.

Vector clock and watermark. Vector clocks and watermarks are widely used techniques to track dependency and maintain consistency in distributed systems [2, 22, 52, 61, 63, 65, 71]. Despite using the same techniques, Mako generally has a different goal and therefore an overall different system design. For example, CURE [2] is a geo-replicated store that uses vector clocks in transaction execution and watermarks to control stable snapshots. It targets transactional causal consistency, while Mako implements serializability, so the goals and techniques are different.

10 Conclusion

This paper presents Mako, a highly-available, fast, and scalable transactional database system, specifically optimized for geo-replication. Mako decouples transaction execution and replication, makes execution speculative, and leverages multi-core machines. Our experimental evaluation shows that Mako outperforms state-of-the-art geo-replicated transactional systems by an order of magnitude in throughput.

Acknowledgments

We thank the anonymous reviewers for suggestions that improved our work. This project was funded in part by NSF awards CNS-2107147, CNS-2321726, CNS-2326576, CNS-2045861, CNS-2321725, CNS-2238768, and CNS-2130590.

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[3] Sergio Almeida, Joao Leitao, and Luis Rodrigues. ChainReaction: a Causal+ Consistent Datastore based on Chain Replication. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2013.

[4] Ahmed Alquraan, Sreeharsha Udayashankar, Virendra Marathe, Bernard Wong, and Samer Al-Kiswany. LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2024.

[5] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. Nimble: Rollback Protection for Confidential Cloud Services. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[6] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: managing datastore locality at scale with Akkio. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[7] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-A Transactional Record Manager for Shared Flash. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[8] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[9] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: it's time for a redesign. *The Proceedings of the VLDB Endowment (PVLDB)*, 2016.

[10] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A distributed in-memory graph database. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2020.

[11] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R K Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[12] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. Morty: Scaling Concurrency Control with Re-Execution. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2023.

[13] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[14] Francesco Calabrese, Mi Diao, Giusy Di Lorenzo, Joseph Ferreira Jr, and Carlo Ratti. Understanding individual mobility patterns from urban sensing data: A mobile phone trace example. *Transportation research part C: emerging technologies*, 2013.

[15] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. Logstore: A cloud-native and multi-tenant log database. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2021.

[16] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[17] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2016.

[18] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin,

Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. TaoBench: An end-to-end benchmark for social network workloads. *The Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[19] Austin T Clements, M Frans Kaashoek, Eddie Kohler, Robert T Morris, and Nickolai Zeldovich. The scalable commutativity rule: designing scalable software for multicore processors. *Communications of the ACM*, 2017.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[21] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[23] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2013.

[24] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *The Proceedings of the VLDB Endowment (PVLDB)*, 2023.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[26] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[27] Tamer Eldeeb, Xincheng Xie, Philip A Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[28] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High Performance Transactions via Early Write Visibility. *The Proceedings of the VLDB Endowment (PVLDB)*, 2017.

[29] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *The Proceedings of the VLDB Endowment (PVLDB)*, 2017.

[30] Hua Fan and Wojciech Golab. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *The Proceedings of the VLDB Endowment (PVLDB)*, 2019.

[31] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *The Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[32] Chathuri Gunawardhana, Manuel Bravo, and Luís ET Rodrigues. Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2017.

[33] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2014.

[34] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2021.

[35] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A Bernstein. Cornus: atomic commit for a cloud DBMS with storage disaggregation. *The Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[36] Theo Haerder and Kurt Rothermel. Concepts for transaction recovery in nested transactions. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 1987.

[37] Jeffrey Helt, Abhinav Sharma, Daniel J. Abadi, Wyatt Lloyd, and Jose M. Faleiro. C5: Cloned Concurrency Control That Always Keeps Up. *The Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[38] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1990.

[39] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *The Proceedings of the VLDB Endowment (PVLDB)*, 2023.

[40] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. KuaFu: Closing the parallelism gap in database replication. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2013.

[41] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *The Proceedings of the VLDB Endowment (PVLDB)*, 2020.

[42] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *The Proceedings of the VLDB Endowment (PVLDB)*, 2020.

[43] Theo Jepsen, Leandro Pacheco de Sousa, Huynh Tu Dang, Fernando Pedone, and Robert Soulé. Optimistic aborts for geo-distributed transactions. *arXiv preprint arXiv:1610.07459*, 2016.

[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2018.

[45] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[46] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[47] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[48] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2021.

[49] Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 1995.

[50] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2016.

[51] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2015.

[52] Masoomeh Javidi Kishi, Sebastiano Peluso, Henry F Korth, and Roberto Palmieri. SSS: scalable key-value store with external consistent and abort-free read-only transactions. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[53] W Kohler, A Shah, and F Raab. Overview of TPC Benchmark C: The Order-Entry Benchmark. *Transaction Processing Performance Council, Technical Report*, 1991.

[54] Ziliang Lai, Hua Fan, Wenchao Zhou, Zhanfeng Ma, Xiang Peng, Feifei Li, and Eric Lo. Knock Out 2PC with Practicality Intact: a High-performance and General Distributed Transaction Protocol. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2023.

[55] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of ACM*

Symposium on Operating Systems Principles (SOSP), 2017.

[56] Junru Li, Youyou Lu, Yiming Zhang, Qing Wang, Zhuo Cheng, Keji Huang, and Jiwu Shu. SwitchTx: scalable in-network coordination for distributed transaction processing. *The Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[57] Yishuai Li, Yunfeng Zhu, Chao Shi, Guanhua Zhang, Jianzhong Wang, and Xiaolu Zhang. Timestamp as a Service, not an Oracle. *The Proceedings of the VLDB Endowment (PVLDB)*, 2024.

[58] Zhongmiao Li, Peter Van Roy, and Paolo Romano. Speculative transaction processing in geo-replicated data stores. Technical report, Instituto Superior Tecnico, Lisboa & INESC-ID, 2017.

[59] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2017.

[60] Gang Liu, Leying Chen, and Shimin Chen. Zen: a high-throughput log-free OLTP engine for non-volatile main memory. *The Proceedings of the VLDB Endowment (PVLDB)*, 2021.

[61] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[62] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[63] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A Fast and Practical Deterministic OLTP Database. *The Proceedings of the VLDB Endowment (PVLDB)*, 2020.

[64] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based commit and replication in distributed OLTP databases. *The Proceedings of the VLDB Endowment (PVLDB)*, 2021.

[65] Umang Mathur and Mahesh Viswanathan. Atomicity checking in linear time using vector clocks. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[66] Alexander Merritt, Ada Gavrilovska, Yuan Chen, and Dejan Milojicic. Concurrent log-structured memory for many-core key-value stores. *The Proceedings of the VLDB Endowment (PVLDB)*, 2018.

[67] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2013.

[68] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[69] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[70] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2015.

[71] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. Detock: High performance multi-region transactions at scale. In *Proceedings of the ACM on Management of Data (SIGMOD)*, 2023.

[72] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[73] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2014.

[74] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 2015.

[75] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, and Mengzhi Wang. Zanzibar: Google's Consistent,

Global Authorization System. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2019.

[76] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 1979.

[77] Karin Petersen, Mike J Spreitzer, Douglas B Terry, Marvin M Theimer, and Alan J Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[78] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. RainBlock: Faster transaction processing in public blockchains. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2021.

[79] Calton Pu, Gail E Kaiser, and Norman C Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1988.

[80] Shujian Qian and Ashvin Goel. Massively Parallel Multi-Versioned Transaction Processing. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[81] Dai Qin, Angela Demke Brown, and Ashvin Goel. Scalable replay-based replication for fast databases. *The Proceedings of the VLDB Endowment (PVLDB)*, 2017.

[82] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2016.

[83] Kun Ren, Dennis Li, and Daniel J Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *The Proceedings of the VLDB Endowment (PVLDB)*, 2019.

[84] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[85] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2022.

[86] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *The Proceedings of the VLDB Endowment (PVLDB)*, 2013.

[87] Dale Skeen. Nonblocking commit protocols. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 1981.

[88] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[89] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2007.

[90] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2020.

[91] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: fast and atomic RDMA-based replication. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2018.

[92] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. An analysis of concurrency control protocols for in-memory databases with CCBench. *The Proceedings of the VLDB Endowment (PVLDB)*, 2020.

[93] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 1979.

[94] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2012.

[95] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[96] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, and

Andy Woods. Enabling the next generation of multi-region applications with cockroachdb. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2022.

[97] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora : Design considerations for high throughput cloud-native relational databases. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2017.

[98] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2018.

[99] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. *The Proceedings of the VLDB Endowment (PVLDB)*, 2017.

[100] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2016.

[101] Jack Waudby, Paul Ezhilchelvan, Isi Mitrani, and Jim Webber. A performance study of epoch-based commit protocols in distributed OLTP databases. In *International Symposium on Reliable Distributed Systems (SRDS)*, 2022.

[102] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2021.

[103] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[104] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[105] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[106] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655*, 2016.

[107] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with RDMA networks. *The Proceedings of the VLDB Endowment (PVLDB)*, 2019.

[108] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the Annual Middleware Conference (Middleware)*, 2015.

[109] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[110] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[111] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[112] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *The Proceedings of the VLDB Endowment (PVLDB)*, 2021.

[113] Zihao Zhang, Huiqi Hu, Xuan Zhou, Yaofeng Tu, Weining Qian, and Aoying Zhou. Fast Commitment for Geo-Distributed Transactions via Decentralized Co-coordinators. *The Proceedings of the VLDB Endowment (PVLDB)*, 2024.

[114] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. Starry: multi-master transaction processing on semi-leader architecture. *The Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[115] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2024.

Appendix

A Proof of Correctness

In this proof, we assume the correctness of exisiting well-known protocols (OCC, Paxos) and mainly aim to prove two properties: commit/rollback atomicity and rollback safety.

Informally, commit/rollback atomicity means if a transaction is committed or rolled back on one shard, all its WriteSet spanning different shards will be committed or rolled back unanimously. There are two cases to discuss. The normal case is that a transaction successfully finishes speculative transaction execution and certification, its WriteSet appears in all relevant shards, and all shards need to unanimously commit or roll back this WriteSet. The abnormal case is that a transaction does not finish speculative transaction execution or certification, e.g., a shard times out waiting for an install call.

The rollback safety means, if a transaction is rolled back, all transactions that depend on this transaction also have to be rolled back. It also implies that if a transaction is replayed or released to client, it cannot be rolled back any more. There are mainly two cases to consider: the transaction and its descendant are in the same epoch, and when they are in different epochs.

Definition 1 (dependency)**.** If transaction T_0 writes to a key and subsequently transaction T_1 reads from the same key, we refer to this as T_1 being read-dependent on T_0. Throughout this proof, the term "dependency" specifically refers to read-dependency.

Definition 2 (writes-*i*)**.** A distributed transaction may update data stored across many shard leaders. Throughout this proof, the term "writes-*i*" refers to the data in the WriteSet for the transaction on shard server-*i*.

Definition 3 (vector clock/vector watermark)**.** Throughout this proof, we consistently denote the vector clocks for transactions T_0 and T_1 as $(p_0, p_1, \ldots, p_{n-1})$ and $(q_0, q_1, \ldots, q_{n-1})$ respectively, and FVW is (w_0,w_1 ... w_{n-1}), where n is the number of shards.

Fact 1. Mako's distributed OCC protocol shares similarities with widely used OCC and 2PC protocols in system designs, and its correctness is straightforward. The core principle of Mako's distributed OCC protocol is that a transaction is considered to have reached a serializable point if it can satisfy two key conditions during the commit phase, consistent with other OCC protocols: (1) All write locks are successfully acquired, and (2) No read items have been modified by other concurrent transactions.

Fact 2. Vector clock of a transaction can reflect the serialization order on shard(s), ensured by our our distributed OCC.

Lemma 1. *If transaction T_1 depends on T_0, then their vector clocks satisfy: $vc_1 \geq vc_0$.*

Proof. If the transaction T_1 depends on T_0, in *GetClock* phase, the vector clock of T_0 must be included in T_1's ReadSet. This allows the Mako to determine T_1's vector clock by selecting the maximum of shard clocks among all vector clocks in its ReadSet, resulting in $p_i \leq q_i$ where $i \in [0, n-1]$. □

Lemma 2. *If T_1 transitively depends on T_0, then $vc_1 \geq vc_0$.*

Proof. The transitivity is obvious. Assume the chain of dependency is $T_1, T_2, \ldots T_k, T_0$. We have $vc_1 \geq vc_2 \geq \ldots \geq vc_k \geq vc_0$. □

Fact 3. For a Paxos stream on the shard leader-*i*, the shard clock-*i* of Paxos entries within this Paxos stream is monotonically increasing within the same epoch, until the end of the epoch.

Lemma 3. *For each shard, w_i in its local view of the vector watermark (w_0, w_1, ..., w_{n-1}) consistently increments, and represents the replication progress of shard leader-i within the same epoch.*

Proof. According to Fact 3, the shard clock-i within the same Paxos stream consistently increases on shard-*i*. Combining this with the fact that w_i is picking up the *minimum* among all shard clock-*i* from the vector clocks of replicated transactions across all Paxos streams on the shard-*i*, we can infer the following. (1) The replication progress (w_i) of shard-*i* is always advancing, and (2) If a transaction updates at least one key on shard-*i* and its shard clock-*i* smaller than w_i, the writes-*i* of this transaction on the shard-*i* has been successfully replicated. □

Lemma 4. *For an epoch e, computing its FVW is a re-entrant and deterministic process. The FVW is the epoch's maximum vector watermark, i.e., for any vector watermark vw in this epoch, $fvw \geq vw$.*

Proof. Given an epoch *e*, the FVW is computed only during the epoch advancement in the failure recovery to ensure its maximum. Once a shard leader receives an epoch advancement request from CM, it needs to address in-flight transactions (*they would be the last transaction for each worker thread within the old epoch*) before advancing to the new epoch:

Case 1 (the good case): All transactions from epoch *e* finished speculative certification or abort before timeout. Then the shard replicates INFs as the end of the epoch. The finalized watermark for this shard is INF.

Case 2 (the abnormal case): Some transactions cannot finish before timeout due to contacting the failed shards or network anomalies. Or all transactions finish but the shard does not successfully replicate INFs. In such cases, at least one Paxos stream of the shard cannot end with INF, either ending with a no-op or with a normal transaction log entry. In both cases, the Paxos stream in this epoch cannot grow any further; its last log entry has the largest clock on this stream (Fact 3). Without loss of generality, assume this clock is the lowest of all streams on this shard. The clock is the shard's watermark in FVW.

In both Case 1 and 2, FVW is computed by choosing for each shard the largest acceptable clock value (the minimum of all streams), which is a fixed value once an epoch ends. Computing FVW is hence a re-entrant and deterministic process, and any other vector watermark in this epoch must be lower than FVW.

□

Fact 4. All of the writes-i of transaction T have the same vector clock.

Lemma 5. *A writes-i of a transaction T is rolled back by shard-i if and only if T's vector clock is not lower than FVW.*

Proof. Given Lemma 4 states that the FVW is a deterministic value for each epoch, if T's vector clock is not lower than FVW, it will be rolled back. Otherwise, T would be replayed (or on the leader becomes visible) and not be rolled back. □

Lemma 6. *If a transaction T partially commits (timeout on some shards), T's vector clock is not lower than FVW.*

Proof. A transaction T partially commits due to timeout on some shard, and will pause the worker thread that processes T on that shard. Assume the Paxos stream this worker threads maps to has its last entry as T'. T' has a lower shard clock than T. In computing the FVW, FVW's w_i must be equal to or lower than than T''s shard clock. This means T cannot have a lower vector clock than FVW and thus T will be rolled back. □

Theorem 1. *(Atomicity) A speculatively certified transaction will either commit or rollback on all relevant shards.*

Proof. Based on Lemma 6, if a transaction partially commits, its vector clock will not be lower than FVW, and hence will be rolled back. Based on Fact 4 and Lemma 5, if a transaction is rolled back, it is rolled back on all relevant shards. □

Lemma 7. *A transaction in a lower epoch cannot depend on any transaction that is in a higher epoch.*

Proof. This is guaranteed by the protocol aborting a transaction if it reads a higher-epoch transaction. □

Lemma 8. *A transaction in a higher epoch cannot depend on any transaction that is or will be rolled back in a lower epoch.*

Proof. This is guaranteed by how our protocol processes the new-epoch transactions:

Case 1: Before FVW is computed, transactions in a new epoch delay accessing versions from the prior epoch e that may be subject to potential rollback.

Case 2: Afterwards, once (e, FVW) is established, transactions in the new epoch only read versions from epoch e defined by the (e, FVW). □

Theorem 2. *(Rollback safety) If a transaction T_0 is rolled back, any transaction T_1 that depends on T_0 will also be rolled back.*

Proof. First, based on Lemma 7, T_1 is either in the same epoch or in a higher epoch than T_0. Further based on Lemma 8, T_1 cannot be a higher epoch, so T_0 and T_1 must be in the same epoch.

Given Lemma 2, T_1 has a greater vector clock than T_0, i.e., $vc_1 \geq vc_0$. Based on Lemma 5, vc_0 is not lower than FVW. Since $vc_1 \geq vc_0$, vc_1 is not lower than FVW. Again based on Lemma 5, T_1 will be rolled back as well.

□

B Shard management in detail

This section provides a detailed explanation of our shard management approach. To enable operations such as shard deletion, addition, and key partition reassignment, we use a management configuration that encapsulates crucial details, such as whether keys remain on the same server (compared to the previous configuration), the total number of shards in the new setup, and the identifiers of those shards.

Two key observations shape our approach: (1) cross-epoch vector clock and vector watermark computations and comparisons are unnecessary in Mako, and (2) transactions in the old epoch are not dependent on those in the new epoch. Once Mako obtains the FVW for the old epoch, the entire database can be treated as a read-only snapshot, serving as the initial state with all vector clocks reset to zero (vector clocks from higher epochs have a higher priority). This still ensures that dependency relationships are preserved, as vector clocks only track dependencies, and vector watermarks track replication progress to prevent data loss.

Building on these observations, our solution treats shard management similarly to failure recovery. The Configuration Manager (CM) oversees the management process by broadcasting a *MIGRATE-START* message containing the new configuration to all shards, including those being deleted or newly added. Upon receiving this message, shard leaders transition to the new epoch and reject any RPCs from the old

epoch or from invalid shards (e.g., those deleted in the new configuration).

Shard leaders then begin executing most transactions speculatively, with two constraints: (1) keys must be below their current view of vector watermarks to avoid potential data loss, and (2) keys scheduled for migration cannot be accessed to ensure safety in the event of a shard failure during the shard management. Since there is no need for all vector clocks from the old epoch to compare with new epoch vector clocks, eliminating concerns about mismatches between vector clocks across epochs.

In the background, shards in the old configuration close the old epoch and exchange shard watermarks to compute a FVW. This process removes the first constraint, and the computed FVW is persisted in the CM for future retrieval, eliminating the need for independent recomputation.

Simultaneously, each shard migrates key-value pairs to other configured shards while retaining the original copies. Once a shard completes its migration task, it reports back to the CM. A background thread reclaims the original copies only *after* the migration tasks for all shards are complete. Shards marked for deletion are only responsible for closing Paxos streams in the old epoch.

When *all* migration tasks are completed, the CM issues a *MIGRATE-END* RPC, lifting the second constraint and enabling full transaction execution under the new configuration. In the event of a shard failure during the migration procedure, the CM initiates a failure recovery procedure, assigning a new epoch. After recovery, the CM can safely restart migration procedure from the beginning.

C Implementations

Multi-version and lazy memory de-allocation. As described in the paper, Mako stores multiple versions of each key to support rollbacks in shard failures. This is implemented as a linked list. Each version contains a pointer to the previous version on this key. The key index stores the pointer to the latest version. We choose this implementation for the multi-version to enable more efficient rollbacks. When rolling back due to shard failures, the operation often requires reclaiming memory for all versions that are incomparable to or above the *FVW* of the old epoch, which can be very expensive. Instead, Mako defers this operation to the future, usually the next time the key is accessed. In our experience, we find this is an engineering effort that can greatly reduce the rollback time. Note that, some unnecessary rollbacks remain unavoidable due to the lack of fine-grained dependency tracking between individual transactions, which would incur high overhead.

Event-driven helper threads. In Mako, the limitation of available CPUs makes it impractical to maintain a large number of busy-polling server threads simultaneously for receiving DPDK messages. As a solution, Mako adopts an event-driven strategy for managing *helper threads* responsible for processing RPC requests from other shard leaders, thereby saving CPU resources. This approach entails temporarily suspending threads and resuming them when required, which regrettably results in additional latency. Despite this trade-off, the event-driven method can efficiently allocate CPU resources and preserve the overall system performance. In our implementation, we use two polling server threads that continuously receive DPDK messages and delegate those messages to the corresponding threads for execution. Mako's RPC relies on eRPC [45] and a private RPC library.

D Pseudocode: transaction execution lifecycle with compression optimization

```
// Execute the transaction in a worker thread.
// Exclude failure recovery logic.

/**** Immutable variable definitions ****/
int nShards;  // Number of shards
int nThreads; // # of worker threads per shard

// Local counter using fetch and add
int localCounter;
// Local shard index
int locShardID;

/**** Mutable variable definitions ****/
int localW; // Local shard watermark
// Replication progress per worker thread
vector<int> replicateProgress(nThreads);

// Size of compressed vector clock (e.g., 10)
int compSize;
// Local view of the compressed vector watermark
vector<int> compVW(compSize);
// Key-value store (simplified as a map)
map<int, ValueType> masstree;

/**** Utils ****/
// Get compressed shard clock index from shard ID
// We assume a simple mapping case
int getCompSIdBySId(int shardId) {
  return shardId / (nShards / compSize);
}

/**** Execute a transaction T0 in Mako ****/
// The worker thread index
int threadID;
// The current epoch for T0
int currentEpoch;

// Execution phase
{
  // Read() optimistically, and buffer writes
  // ...
}

// Commit phase
{
  {
    // Phase1: lock keys in writeset
    // ...
  }

  // Get the vector clock for T0
  vector<int> compVC = getClock();

  {
    // Phase2: Validate keys in readset
    // ...
  }

  // Replicate transaction logs (batch for the
      optimization)
  // Inovke the callback func once a log is durable
  asyncPaxosRep(/*resultant values of transactions*/,
                /*compVC*/,
                /*threadID*/,
                /*locShardID*/,
                callbackAsyncPaxosRep);

  // Phase3: Install & Release locks
  Install(writeSet, currentEpoch, compVC);
}

// Read from all relevant shards according to keys
// T0 always reads latest version if no failures
tuple<int,vector<int>,ValueType>
  Read(key, currentEpoch) {
    _epoch, _compVC, _value = masstree[key][0];
    return {_epoch, _compVC, _value};
}


// Get compressed vector clock for the transaction
vector<int> getClock() {
  vector<int> _compVC(compSize);

  // Merge compressed vector clocks in ReadSet
  for (auto _cvc : readSet) {
    for (int i=0; i<compSize; i++) {
       _compVC[i] = MAX(_compVC[i], _cvc[i])
    }
  }

  // Broadcast a RPC to each remote shard once in
      WriteSet to increment remotes' local counter
  for (auto shardId: writeSet) {
    compSId = getCompSIdBySId(shardId);
    sclock = remoteIncrement(shardId);
    _compVC[compSId]= MAX(_compVC[compSId], sclock);
  }

  return _compVC;
}

void Install(writeSet, currentEpoch, compVC) {
  for (auto k, v: writeSet) {
    {// Execute on all relevant shards
      masstree[k].insert(0,(currentEpoch, compVC, v));
      compSId = getCompSIdBySId(locShardID);
      // Ensure subsequent transactions observe a
          larger value
      int delta = compVC[compSId] - localCounter;
      if (delta > 0)
         localCounter.fetch_add(delta);
    }
  }
}

void callbackAsyncPaxosRep(int threadID, vector<int>
    compVC, int locShardID, const string&
    resultantValue) {
  compSId = getCompSIdBySId(locShardID);
  replicateProgress[threadID] = compVC[compSId];
  localW = MIN(replicateProgress);
  compVW[compSId] = MIN(compVW[compSId], localW);

  {
    // Exchange shard watermarks periodically to
        update compVW
    // ...
  }

  recvQueues.push_back((resultantValue, compVC));

  // Replay on shard followers
  while (!recvQueues.empty()) {
    bool safeToReplay = true;
    for (int i=0; i<compSize; i++) {
      if (recvQueues.front().compVC[i] > compVW[i]) {
        safeToReplay = false;
        break;
      }
    }

    if (!safeToReplay) break;

    // It is safe to return back the client and replay
    replay(recvQueues.front());
    recvQueues.pop_front();
  }
}
```





Quake: Adaptive Indexing for Vector Search

Jason Mohoney
University of Wisconsin-Madison

Devesh Sarda
University of Wisconsin-Madison

Mengze Tang
University of Wisconsin-Madison

Shihabur Rahman Chowdhury
Apple

Anil Pacaci
Apple

Ihab F. Ilyas
University of Waterloo

Theodoros Rekatsinas
Apple

Shivaram Venkataraman
University of Wisconsin-Madison

Abstract

Vector search, the task of finding the k-nearest neighbors of a query vector against a database of high-dimensional vectors, underpins many machine learning applications, including retrieval-augmented generation, recommendation systems, and information retrieval. However, existing approximate nearest neighbor (ANN) methods perform poorly under dynamic and skewed workloads where data distributions evolve. We introduce Quake, an adaptive indexing system that maintains low latency and high recall in such environments. Quake employs a multi-level partitioning scheme that adjusts to updates and changing access patterns, guided by a cost model that predicts query latency based on partition sizes and access frequencies. Quake also dynamically sets query execution parameters to meet recall targets using a novel recall estimation model. Furthermore, Quake utilizes NUMA-aware intra-query parallelism for improved memory bandwidth utilization during search. To evaluate Quake, we prepare a Wikipedia vector search workload and develop a workload generator to create vector search workloads with configurable access patterns. Our evaluation shows that on dynamic workloads, Quake achieves query latency reductions of 1.5–38× and update latency reductions of 4.5–126× compared to state-of-the-art indexes such as SVS, DiskANN, HNSW, and SCANN.

1 Introduction

Vector search, the task of finding the k-nearest neighbors (KNN) of a query vector against a database of high-dimensional vectors, is fundamental to modern machine learning based search [9, 11, 12, 36] and recommendation systems [21, 22, 32, 33, 43]. In these applications, a vector represents an item in a metric space, and the distance between vectors reflects semantic similarity. However, performing exact KNN search becomes computationally infeasible on large datasets due to the high dimensionality and volume of data.

To address this challenge, practitioners use approximate nearest neighbor (ANN) indexes, which trade off a controlled amount of search accuracy (recall) for significant reductions in latency. Among these, two broad classes dominate in practice: graph-based and partitioned indexes, each with distinct performance characteristics under dynamic workloads.

Maintaining low latency, high recall vector search under **dynamic and skewed workloads** remains a significant challenge for existing indexes. Real-world applications often exhibit non-uniform query distributions and evolving data. For example, in an example Wikipedia search application, popular pages like *Lionel Messi* or *LeBron James* receive disproportionately more queries, resulting in *skewed read patterns*. Additionally, pages are frequently added, updated, or deleted, causing *skewed update patterns* that change over time [6]. These factors degrade the performance of existing indexes, leading to increased query latency and reduced recall.

Graph-based indexes, such as HNSW [24], DiskANN [38, 39], and SVS [5] construct a proximity graph where each node (vector) is connected to its approximate neighbors. Queries traverse the graph to find approximate nearest neighbors, typically achieving high recall with low latency. However, these indexes face challenges with dynamic workloads because updating the graph structure to accommodate frequent insertions and deletions is computationally intensive [45], due to the random access patterns involved in graph traversal and modification.

Partitioned indexes, such as SCANN [10, 40], SPANN [7, 45], and Faiss-IVF [8], partition the vectors using a clustering algorithm (e.g k-means). Queries are processed by scanning a subset of partitions, balancing recall and latency by adjusting the number of partitions scanned (denoted as *nprobe*). While attractive due to their simplicity, partitioned indexes face a significant search latency gap when compared with graph indexes. For example, on the MSTURING10M benchmark [2], we found Faiss-IVF takes 44ms per search query while Faiss-HNSW takes only 6.8ms. On the other hand, supporting updates in partitioned indexes is less expensive than for graph indexes, as the index structure needs minimal modification when adding or removing vectors. But, existing approaches struggle with dynamic and skewed workloads because they do not adapt to changing access patterns, leading to *imbalanced* partitions that degrade query latency.

Recent work has been proposed to resolve imbalances in dynamic workloads by splitting and reclustering imbalanced partitions [6, 45], however, we find these methods degrade recall as nprobe needs to change as the index structure changes.

In this work, we study the problem of minimizing query latency to meet a fixed recall target for dynamic vector search workloads, where both the queries and the base vectors can change over time. To address this problem, we develop **Quake**, a partitioned index for ANN search that minimizes query latency by adapting the index structure to the workload. Quake's two primary algorithmic contributions are:

First, Quake employs an **adaptive hierarchical partitioning** scheme that modifies the partitioning by minimizing the cost (derived from a proposed **cost model**) of a query. The cost model tracks partition sizes and access frequencies as the workload is processed and determines which partitions are most negatively contributing to overall query latency. Once identified, we split or merge these partitions based on expected cost reduction derived from our proposed cost model. We also demonstrate that our maintenance procedure is stable and converges to a local minimum of the cost model.

Second, we design an **adaptive partition scanning** scheme that adjusts the number of partitions scanned on-the-fly to meet recall targets for individual queries. We do this by maintaining a recall estimate during query processing based on A) the geometry of the partitioning and B) intermediate results of the query, and once the estimate exceeds the recall target, query processing terminates and the results are returned.

Furthermore, Quake utilizes NUMA-aware parallelism to maximize memory bandwidth usage on multi-core machines.

It is a significant challenge to evaluate indexing approaches due to the lack of availability of benchmarks for online vector search. To address this challenge and comprehensively evaluate our approach, we A) prepare a **Wikipedia vector search workload** derived from publicly available query and update patterns of Wikipedia pages and B) develop a **workload generator** for creating workloads with configurable query and update patterns. We will publicly release the Wikipedia workload and workload generator as evaluation tools for the community to use. Using these, we conduct a comprehensive evaluation of Quake in comparison to seven baseline approaches.

1. Quake achieves the lowest search time across all dynamic workloads compared to state-of-the-art graph indexes, with 1.5-13× lower search latency than HNSW, DiskANN, and SVS while having 18-126× lower update latency.
2. We also find that APS matches the nprobe of an oracle across recall targets on SIFT1M, with only a 17-29% increase in latency relative to the oracle.
3. APS performs on-par or better than existing early termination methods [7, 18, 48] and requires no offline tuning.
4. Quake's NUMA-aware query processing exhibits linear scalability and high memory bandwidth utilization on the MSTURING100M dataset. Quake achieves 20× and 4× lower query latency compared to single-threaded and non-NUMA aware configurations, respectively.

2 Motivation and Challenges

Efficient vector search is critical for large-scale systems used in recommendation, semantic search, and information retrieval. These applications demand the ability to process a high volume of nearest neighbor queries with low latency, even as the underlying data evolves. To meet these requirements, vector databases—such as Milvus [42], Pinecone [3], AnalyticDB-V [44], VBASE [47], and Qdrant [1]—utilize specialized vector indexes (e.g., Faiss-IVF, HNSW, Vamana) that support fast approximate nearest neighbor (ANN) queries. However, serving these dynamic workloads introduces significant challenges in maintaining query performance and accuracy as data and query patterns shift over time.

2.1 Vector Search Workload

A vector search workload is a continuous, evolving stream of **queries** and **updates**:

- **Queries:** Given a query vector q, the goal is to find the top-k nearest neighbors in a set $\mathbf{X}$. Exact linear search is too slow for large, high-dimensional datasets, so ANN indexes are used. These indexes approximate nearest neighbors with controlled recall to lower latency by orders of magnitude.
- **Updates:** The dataset evolves over time. Insertions add new vectors representing fresh content (e.g., new products, trending news articles), and deletions remove outdated entries. Typically, updates are applied in a batched fashion.

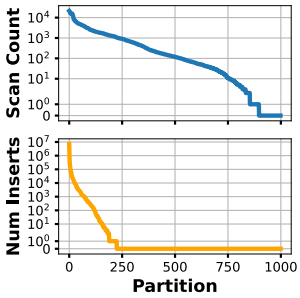
Recall@k is the standard metric for accuracy, defined as: $\frac{|\mathbf{G} \cap \mathbf{R}|}{k}$ where $\mathbf{R}$ is the vectors returned by the approximate search, and $\mathbf{G}$ is the ground truth set. Maintaining a consistent recall target (e.g., $> 90\%$) and low latency (e.g., milliseconds per query) as both data and query patterns shift is a key challenge. The complexity of these workloads stems from their inherently dynamic and skewed nature, which few existing indexing methods handle gracefully.

2.2 Why Real-World Workloads are Hard

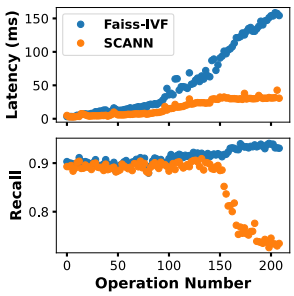
Skewed Read Patterns In practice, user queries concentrate on popular items. For example, queries against a Wikipedia-derived dataset tend to focus on a small subset of entities at any given time. As a result, certain partitions or graph regions of the index are accessed disproportionately often.

Skewed Write Patterns Insertions and deletions are also rarely uniform. New data often arrives in bursts—e.g., new Wikipedia pages added monthly, new products introduced ahead of a shopping season, or newly relevant embeddings generated by continuously updated language models.

Real-World Example: Wikipedia-12M In our evaluation, we prepared WIKIPEDIA-12M, a workload based on a subset of Wikipedia articles derived from publicly available monthly pageview statistics [4]. Over 103 months, the dataset grows from millions to tens of millions of vectors. Popular articles



(a) Read (top) and write skew.



(b) Query performance.

Figure 1: Skewed access patterns of Faiss-IVF index partitions in the WIKIPEDIA-12M workload and their effect on query performance for Faiss-IVF and SCANN

Table 1: Comparison of updatable vector indexes. **Tuning**: Requires manual parameter tuning in indexing/query processing. **Maintenance**: Modifies index with incremental updates. **Adaptive**: Uses query information to inform indexing.

Method	Tuning	Maint.	Adaptive
Quake **(Ours)**	✗	✓	✓
Faiss-IVF [8]	✓	✗	✗
DeDrift [6]	✓	✓	✗
SpFresh [45]	✓	✓	✗
SCANN [10,40]	✓	✓	✗
DiskANN [31,38]	✓	✓	✗
Faiss-HNSW [24]	✓	✗	✗
SVS [5]	✓	✓	✗

dominate query traffic, while embeddings of newly created pages accumulate in certain regions of the embedding space. This workload shows read skew and write skew, as evidenced by Figure 1a, reads and writes predominantly affect a small portion of the index.

2.3 Shortcomings of Existing Approaches

Existing indexes were often developed and evaluated under assumptions of static data distributions; conditions not met in real-world use cases. Table 1 compares a range of state-of-the-art vector indexes. Although widely adopted in vector databases, none fully solve the problem of maintaining low-latency, high-recall search under dynamic, skewed workloads without constant manual intervention or offline tuning.

Graph Indexes Graph-based index systems, such as HNSW [24], DiskANN [31, 38], and SVS [5] construct a proximity graph where each node represents a vector connected to its approximate neighbors. These indexes achieve high recall with low latency in static settings by efficiently traversing the graph to locate nearest neighbors using a process known as *greedy traversal*. However, maintaining the graph structure under frequent updates is computationally intensive, as each update may require rewiring multiple edges to preserve graph connectivity and proximity properties. Our evaluation (Table 1) shows that update latency can be multiple orders of magnitude higher than partitioned indexes.

Partitioned Indexes Partitioned indexes such as Faiss-IVF [8], SCANN [10, 40], and SpFresh [45] divide the vector space into disjoint partitions using a clustering algorithm such as k-means. Queries are processed by scanning a subset of partitions to retrieve approximate nearest neighbors. Partitioned indexes are more update-friendly than graph-based method since insertions and deletions leads to sequential access. For write skewed workloads some partitions become significantly larger, degrading query latency, this can be exacerbated by read skew if large partitions are also more frequently accessed ("hot partitions"). Query processing is *memory-bound*, as achieving high recall requires scanning many megabytes of data across multiple partitions, for example reaching a recall target of 90% on the MSTURING100M dataset requires each query to scan 1GB of vectors. Moreover, most partitioned indexes use a fixed number of partitions to probe (*nprobe*), which does not adapt to changing data distributions or query patterns, leading to either insufficient recall or excessive data scanning. The challenges yield subpar performance for partitioned indexes on real-world workloads. For example, Figure 1b shows the degradation of latency and recall over time when using Faiss-IVF and SCANN with a fixed nprobe on WIKIPEDIA-12M (workload details in Section 7).

Early Termination Early-termination methods have been proposed to reduce query latency or meet recall targets in partitioned indexes by dynamically adjusting the number of partitions scanned per query. **SPANN** [7] applies a simple rule: it prunes partitions once the centroid distance exceeds a user-tuned threshold relative to the closest centroid. **LAET** [18] is a learning-based approach that predicts the required *nprobe* per query using a trained model, but still requires dataset-specific training and calibration for each recall target. **Auncel** [48] uses a geometric model to estimate when recall for a given query, setting *nprobe* per query, but its conservative estimation leads to substantial overshooting of the recall target (See Figure 13 in [48]). All three methods require tuning or calibration and do not adapt to changes in the index structure or data distribution.

2.4 Technical Challenges for Partitioned Indexes

The following technical challenges are yet to be solved by existing partitioned indexes

1. **Adaptation to Queries** Query adaptivity is overlooked by existing partitioned index approaches and exhibits an opportunity for optimization, particularly for maintaining hot partitions induced by read skew.
2. **Online Adjustment of Nprobe** As the index structure and data change, partitioned indexes need to adjust the number of partitions scanned or recall will suffer. Existing early termination works are insufficient as they assume a static index and require retuning as the index and data change.

3. **Performance Gap with Graph Indexes** Standard partitioned indexes such as Faiss-IVF are memory bound, and exhibit an order of magnitude higher query latency in comparison than graph indexes.

Quake is our solution to these technical challenges. Quake A) adapts the index structure to queries by utilizing maintenance that minimizes a cost model for query latency, B) using a recall estimation model, Quake individually sets nprobe for queries to meet recall targets as the index structure changes, and C) uses NUMA-aware parallelism in order to saturate memory bandwidth during query processing, closing the performance gap with graph indexes. We next briefly discuss other related work before covering Quake in detail.

3 Solution Overview

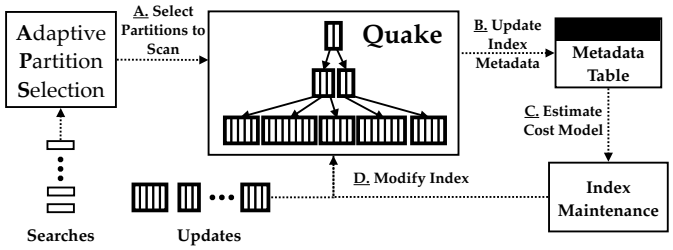


Figure 2: Quake Architecture Diagram. Search queries use Adaptive Partition Selection (APS) to determine which partitions to scan (**A**). Scanning partitions modifies access patterns of the index, tracked in the metadata table (**B**). A cost model is used to determine which maintenance actions to take (**C**) where the chosen maintenance actions modify the index (**D**). This process operates in a continuous online fashion as search and update queries (inserts/deletes) are issued to the index.

Index Structure Quake organizes the vectors in a multi-level index, where each level is a partitioned index similar to Faiss-IVF [8]. The lowest level in the index is constructed by organizing the vectors into disjoint partitions (using k-means clustering) where each partition has a representative centroid. These centroids can be further partitioned in a similar manner to create additional levels in the index. Search queries scan the index in a top-down fashion, finding the nearest centroids at each level to determine the partitions to scan in the next level. Partitions in the lowest level contain the actual vectors and subsets of these partitions are scanned to return the k-nearest neighbors. Utilizing a multi-level design enables us to employ fine-grained partitioning of vectors at large scale (shown to improve search quality [7]), while mitigating the high cost of scanning centroids.

Adaptive Incremental Maintenance Inserts and deletes modify the Quake data structure by appending vectors to and removing vectors from index partitions. Insertions traverse the index structure top-down to find the nearest partition in the lowest level to the inserted vector and append to that partition. Deletes use a map to find the partition containing the vector to be deleted and the vector is removed from the partition with immediate compaction. As demonstrated in Section 2, modifications can negatively affect index performance over time, requiring maintenance (Figure 1b). Quake uses the following maintenance actions in order to minimize query latency:

1. **Split Partition:** Uses k-means clustering to split a partition into two, removing the old partition and its centroid and adding two new partitions and centroids. To mitigate potential overlap due to the new partitions, we perform additional iterations of k-means clustering over the partitions neighboring the split partitions (by centroid distance).
2. **Merge Partition:** Removes a partition and its centroid, reassigning the vectors of the removed partition to the remaining partitions in the index.
3. **Add Level:** Adds a level of partitioning to the index by partitioning the current top-level using k-means clustering.
4. **Remove Level:** Removes current top-level and merges the partitions in the next level.

Quake uses a cost model that estimates query latency to determine if maintenance actions should be taken and which partitions to apply them to. The cost model is a function of partition access patterns and sizes to determine which partitions are contributing most to the overall query latency. We check for maintenance after each operation by evaluating the cost model, but the maintenance frequency is configurable. Partitions with the largest cost contribution are considered for split or deletion. Intuitively, frequently accessed and/or large partitions are split and infrequently accessed and/or small partitions are merged as they do not justify the overhead of maintaining a centroid. See Section 4 for details on the cost model and maintenance methodology.

Adaptive Partition Scanning In order to determine the number of partitions a search query should scan to reach a given recall target, we apply *Adaptive Partition Scanning* (APS) at each level of the index. APS solves a critical problem for partitioned indexes when applied to dynamic workloads: as the number and contents of partitions change, the number of partitions scanned needs to change, otherwise recall will degrade (Figure 1b). APS maintains a recall estimator based on the intermediate top-k results of the query and the geometry of neighboring partitions. As more partitions are scanned, the intermediate results and recall model are updated and when the recall estimate exceeds the target recall, the results are returned. To mitigate overheads introduced by the estimator, we use pre-computation of expensive geometric functions and only update the estimate when the intermediate results have changed significantly. APS supports both Euclidean and inner-product distance metrics. We cover APS in Section 5.

NUMA-Aware Query Processing Modern multi-core servers often use Non-Uniform Memory Access (NUMA) architectures, where memory close to a processor's local node is faster to access than remote memory. Quake is designed to capitalize on this heterogeneous memory. It distributes

index partitions across NUMA nodes. To minimize remote memory access, Quake employs affinity-based scheduling, and supports work stealing within a NUMA node to mitigate workload imbalances. By co-locating computation with the relevant data, Quake reduces remote memory accesses, saturates memory bandwidth, and thus lowers query latency. See Section 6 for details on Quake's NUMA-aware optimizations.

4 Adaptive Incremental Maintenance

We present our adaptive maintenance methodology, beginning with a cost model that estimates each partition's contribution to query latency and guides maintenance decisions. Next, we describe the available maintenance actions, analyzing their impact on the cost model. We then detail the multi-stage decision workflow that prioritizes beneficial actions and conclude with a concrete example.

4.1 Cost Model

The cost model estimates the query latency contributed by each partition, in the index. Estimating the per-partition latency contribution enables targeted maintenance to the partitions most affecting query performance.

Partition Properties Consider an index with L levels, numbered $l = 0, 1, \ldots, L-1$. Level l contains N_l partitions. The base level corresponds to $l = 0$ and contains partitions of the original dataset vectors. Higher levels contain partitions of centroid vectors that summarize the partitions in the level below. At the top level, $l = L-1$, there is a single partition containing top-level centroids.

Each partition j at level l has a size s_{lj} (the number of vectors it contains) and an access frequency $A_{l,j} \in [0.0, 1.0]$. $A_{l,j}$ denotes the fraction of queries, measured in a sliding window W, that scan the partition j at level l. The cost model is primarily driven by these sizes and access frequencies.

Partition Cost A partition (l, j) contributes latency proportional to its size and how frequently it is accessed. Let $\lambda(s)$ be the latency function for scanning s vectors. We measure $\lambda(s)$ through offline profiling. The cost of partition (l, j) is:

$$C_{lj} = A_{lj} \cdot \lambda(s_{lj}) \tag{1}$$

Total Cost The overall query latency (cost) estimate is the sum across all levels and partitions

$$C = \sum_{l=0}^{L-1} \sum_{j=0}^{N_l-1} A_{lj} \cdot \lambda(s_{lj}) \tag{2}$$

Interpretation The cost model reflects the relationship between partition size, access frequency, and query latency. The fundamental trade-off that needs to be balanced is the number and size of partitions. Larger partitions require more time to scan, increasing latency, but reducing the total number of partitions and the overhead of scanning centroids. Conversely, smaller, fine-grained partitions reduce the number of vectors needed to scan to reach a high recall but increase the overhead of scanning centroids. The model further captures that frequently accessed partitions dominate the total cost, motivating targeted maintenance actions to balance these trade-offs.

Guiding Maintenance Decisions Maintenance actions such as splitting or deleting aim to reduce the total cost C. Each action is evaluated based on its predicted change in cost:

$$\Delta C = C_{\text{after}} - C_{\text{before}} \tag{3}$$

where C_{before} and C_{after} are the total costs before and after the action, respectively. Actions are applied only if $\Delta C < -\tau$, where τ is a non-negative tunable threshold, ensuring monotonic improvement in query performance. By focusing on reducing C, the index is dynamically restructured to maintain efficient query performance under varying workloads.

4.2 Conducting Maintenance

Maintenance at each level of the index proceeds in three phases—*estimate*, *verify*, and *commit / reject*. We first list the available actions, then derive their cost deltas, describe the workflow, and finally walk through a concrete example.

4.2.1 Maintenance Actions

To minimize query latency, Quake employs a series of maintenance actions that dynamically adjust the index structure in response to evolving workloads. Here we define the maintenance actions and then analyze the impact of each maintenance action on the overall cost model.

Split Partition If a partition (l, j) is too large or frequently accessed, we consider splitting it into two partitions (l, j_L) and (l, j_R). We apply k-means clustering within that partition, forming two smaller partitions with their own centroids. The original partition is removed and its vectors are reassigned. A subsequent *partition refinement* step adjusts vector assignments to ensure minimal overlap and balanced partition sizes.

Partition Refinement After a split, refinement uses k-means (seeded by current centroids) on nearby partitions to mitigate overlap and ensure that each vector is assigned to its most representative partition. Nearby partitions are determined by finding the r_f nearest centroids to the split centroids, where r_f is a tunable parameter (typically between 10 and 100). This is a generalization of the reassignment procedure used in SpFresh/LIRE [45], using additional rounds of k-means prior to reassignment, and has been applied successfully by recent index maintenance works [23] and [25]. Refinement avoids performance degradation by mitigating overlap and ensuring vectors are assigned to their most representative partition.

Merge Partition If a partition is rarely accessed and below a minimum size threshold, we consider deleting it to remove the cost of maintaining its centroid. After deletion, the vectors are reassigned to their respective nearest existing partitions. This can reduce total cost by removing a low-benefit partition, although the reassignment may increase the size (and thus cost) of other partitions, and therefore careful consideration is needed before conducting a merge.

Adding and Removing Levels If the number of centroids in the top level grows beyond a threshold, we add a new top level by clustering those centroids. Conversely, if the top level becomes too sparse (below a configured lower threshold), we remove the top level and merge its centroids in the level below. Both actions help maintain hierarchy balance and control centroid-scanning overhead. We defer discussion of this to our technical report [27].

4.2.2 Cost Deltas

The maintenance loop treats every candidate action as a *proposed edit* to the index and scores it by the change it would induce in the total cost (Eq. (3)).

We tentatively accept an action whenever $\Delta C < -\tau$. Below we give the exact ΔC formulas for the primary maintenance actions: *split* and *merge*. Full derivations are in the technical report [27]; here we show only the final expressions.

Exact Split Delta Splitting a hot or oversized partition (l, j) into children (l, j_L) and (l, j_R) inserts one new centroid at the parent level, changing the overhead by $\Delta O^+ = \lambda(N_l + 1) - \lambda(N_l)$. The resulting cost difference is

$$\begin{aligned}\Delta\text{Split}_{l,j} = \underbrace{\Delta O^+}_{\text{new centroid}} - A_{l,j}\,\lambda(s_{l,j}) \\ + A_{l,j_L}\,\lambda(s_{l,j_L}) + A_{l,j_R}\,\lambda(s_{l,j_R})\end{aligned} \tag{4}$$

where the first term pays for the extra centroid, the second removes the old scan cost, and the last two add the costs of scanning the new, smaller partitions. Note that we do not explicitly model the effect of refinement, as refinement does not change the number of partitions. Its impact is captured automatically as statistics are collected from future queries, so we omit it from the Δ-formula and let later maintenance iterations adjust if necessary.

Exact Merge Delta Deleting a cold, tiny partition (l, j) removes a centroid $(\Delta O^- = \lambda(N_l - 1) - \lambda(N_l))$ and redistributes its vectors to a receiver set $R_{l,j}$. Let Δs_m and ΔA_m be the resulting size and frequency bumps for each receiver m. Then

$$\begin{aligned}\Delta\text{Merge}_{l,j} = \Delta O^- - A_{l,j}\,\lambda(s_{l,j}) \\ + \sum_{m \in R_{l,j}} \left[(A_m + \Delta A_m)\,\lambda(s_m + \Delta s_m) - A_m\,\lambda(s_m)\right]\end{aligned} \tag{5}$$

captures both the benefit of deleting the partition and the penalty of swelling its neighbors.

Estimating Deltas At decision time we do not yet know the post-action quantities $\{s_{l,j_L}, A_{l,j_L}, \dots\}$ or the true $\Delta s_m, \Delta A_m$. We therefore use a lightweight *estimate* based on two assumptions: 1) **Balanced Split:** $s_{l,j_L} \approx s_{l,j_R} \approx \frac{s_{l,j}}{2}$, and 2) **Proportional-Access Scaling**: each child inherits a fixed fraction α of the parent's frequency.

Under these assumptions the split estimate becomes

$$\begin{aligned}\Delta'\text{Split}_{l,j} = \Delta O^+ - A_{l,j}\,\lambda(s_{l,j}) \\ + 2\alpha A_{l,j}\,\lambda\left(\tfrac{s_{l,j}}{2}\right).\end{aligned} \tag{6}$$

and the analogous merge estimate, derived with a uniform redistribution assumption, is located in the technical report [27]. Immediately after a tentative action we measure the *actual* sizes (and, for merges, the exact receiving partitions) and re-evaluate Eqs. (4) or (5). If the recomputed gain is still below $-\tau$ the action is committed; otherwise it is rolled back (§4.2.3). This "estimate-then-verify" strategy is crucial for ensuring monotonic cost improvement.

4.2.3 Decision Workflow

Maintenance is a *bottom-up* pass over the hierarchy. Each level executes the five stages below starting from the base level. This workflow is triggered by the user. An avenue for future work is to develop scheduling policies to call this workflow and limit its scope. In our evaluation, we trigger maintenance after a set amount of queries have been run.

Stage 0: Track Statistics At the end of each query batch we update, for every partition (l, j): (i) size $s_{l,j}$, (ii) access count over the sliding window of queries W, giving $A_{l,j} = \text{hits}(l, j)/|W|$. These values are inputs of the cost model.

Stage 1: Estimate For the current level l compute the estimate Δ' (§4.2.2) of splitting and deleting for every partition. Tentatively apply any action with $\Delta' < -\tau$.

Stage 2: Verify Immediately after performing a tentative action, we measure the actual resulting partition sizes (and the exact receiver partitions for merges). We recompute the cost delta using these known values but retain the original frequency assumptions from Stage 1.

Stage 3: Commit / Reject

$$\Delta < -\tau \;\rightarrow\; \textit{commit}, \quad \Delta \geq -\tau \;\rightarrow\; \textit{reject}.$$

Rejection discards the action and keeps the previous state of the partition(s), in order to prevent cost increases.

Stage 4: Propagate Upward. Repeat Stages 1-3 on the next level $l+1$,

Safety: Because every level enforces the same $\Delta < -\tau$ guard, total cost across *all* levels monotonically decreases and the hierarchy converges to a stable state under a fixed workload distribution (proof in technical report [27]).

4.2.4 Example Maintenance Workflow

Below we walk through the *estimate → verify → commit / reject* loop for two example partitions and show how an imbalanced split is automatically rejected to prevent accidental cost increases.

Set-up: Consider partitions P_1 and P_2 with identical size and access frequency, where both contain $s = 500$ vectors and appear in $A = 0.10$ of queries. From profiling we observe non-linear [1] scan latencies for the following partition

[1] Scan latency is non-linear w.r.t. size due to top-k sorting overhead.

sizes: $\lambda(50) = 250\mu s$, $\lambda(250) = 550\mu s$, $\lambda(450) = 1050\mu s$, $\lambda(500) = 1200\mu s$ Adding a centroid costs $\Delta O^+ = 60\mu s$. We use a decision threshold of $\tau = 4\mu s$ and $\alpha = .5$

1. **Estimate** For P_1 and P_2 the estimate assumes a balanced 250/250 split and $\alpha = 0.5$ traffic per child:

$$C_{\text{before}} = 0.10 \times 1200 = 120\mu s,$$
$$C_{\text{est}} = 0.05 \times (550 + 550) = 55\mu s,$$
$$\Delta' = 60 - 120 + 55 = -5\mu s.$$

   Because $\Delta' < -\tau$, *both P_1 and P_2 are tentatively split*.
2. **Verify** After splitting we see that P_1 has a 250/250 split, but P_2 comes out 450/50:

$$C_{\text{verify}}(P_2) = 0.05 \times (1050 + 250) = 65\mu s,$$
$$\Delta(P_2) = 60 - 120 + 65 = +5\mu s.$$

3. **Decide**
   - P_1: *commit* because $\Delta = -5\mu s < -\tau = -4\mu s$
   - P_2: *reject* because $\Delta = +5\mu s > -\tau = -4\mu s$

   The verify step therefore blocks an imbalanced split that would otherwise *increase* query latency.

5 Adaptive Partition Scanning (APS)

Adaptive Partition Scanning (APS) dynamically determines the number of partitions to scan per query to achieve a specified recall target τ_R with minimal latency. APS adapts to evolving workloads and changing index structures, making it particularly effective in dynamic data settings. We first introduce the geometric model underlying APS, followed by a detailed description of the scanning algorithm, and conclude with key performance optimizations. We apply APS at each level of the index independently. For clarity we focus on the Euclidean distance, we discuss inner product metrics in the technical report [27].

Geometric Model To estimate the probability that each partition contains one of the query's k nearest neighbors, APS uses a geometric interpretation. Given query $\mathbf{q}$ and ρ the distance to the k-th nearest neighbor, consider the hypersphere $\mathcal{B}(\mathbf{q}, \rho)$. Under a uniform-density assumption, the fraction of this sphere's volume intersecting partition $\mathcal{P}_i$ estimates the probability that $\mathcal{P}_i$ holds a nearest neighbor:

$$p_i = \frac{\text{Vol}\big(\mathcal{B}(\mathbf{q}, \rho) \cap \mathcal{P}_i\big)}{\text{Vol}\big(\mathcal{B}(\mathbf{q}, \rho)\big)}, \tag{7}$$

Because we do not know the true distance of the k-th nearest neighbor apriori, we set ρ to the current k-th nearest neighbor observed and update it online as partitions are scanned.

Intersection Volume Approximation Exact computation of intersection volumes between a sphere and high-dimensional Voronoi partition boundaries is infeasible, as partitions are intersections of multiple half-spaces. Instead, we approximate each partition as a single half-space, defined by the perpendicular bisector between the query's nearest centroid $\mathbf{c}_0$ and each neighboring centroid $\mathbf{c}_i$. This simplification results in a hyperspherical cap whose volume v_i has a closed-form expression via the regularized incomplete beta function [16, 19] (see technical report [27]).

Nearest Partition Volume Approximation The half-space approximation is invalid for the nearest partition $\mathcal{P}_0$, since the query lies within it. Instead, we first compute hyperspherical cap volumes v_j for the remaining $M-1$ candidate partitions and normalize these so that $\sum_{j=1}^{M-1} v_j = 1$. The probability p_0 that no neighbor is located outside $\mathcal{P}_0$ is:

$$p_0 = \prod_{j=1}^{M-1} (1 - v_j), \tag{8}$$

with the remaining probability distributed proportionally among other partitions according to their volumes v_i:

$$p_i = (1 - p_0)\, v_i. \tag{9}$$

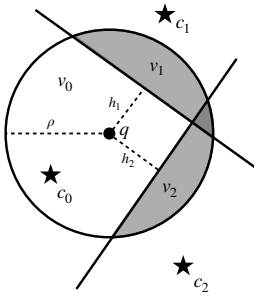


Figure 3: The query hypersphere (centered at $\mathbf{q}$ with radius ρ) intersecting partition boundaries. The intersection volumes v_1 and v_2 correspond to the probability of finding a nearest neighbor in partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively.

5.1 APS algorithm

Algorithm 1 details the APS procedure. Given query $\mathbf{q}$, recall target τ_R, and the initial candidate fraction f_M:

1. Scan partition $\mathcal{P}_0$, initializing the query radius ρ.
2. Compute probabilities p_i for each remaining candidate partitions based on radius ρ.
3. Iteratively scan partitions in descending probability order until cumulative recall exceeds target τ_R, updating radius ρ and recomputing probabilities whenever ρ shrinks significantly (beyond threshold τ_ρ).

This process is conducted at each level of the index. To avoid propagating errors from searching higher levels, we fix the recall target to 99% for the higher levels (Table 6).

Performance Optimizations APS incorporates two optimizations to minimize computational overhead. First, it precomputes values of the regularized incomplete beta function at 1024 evenly spaced points in $[0, 1]$ and linearly interpolates

Algorithm 1 Adaptive Partition Scanning (APS)

Input: query $\mathbf{q}$, centroids $\mathcal{C}$, recall target τ_R, initial candidate fraction f_M, recompute threshold τ_ρ, k
Output: k nearest neighbors of $\mathbf{q}$

$R \leftarrow$ empty max-heap of size k
$\mathcal{S} \leftarrow f_M * N_l$ nearest centroids to $\mathbf{q}$
scan $\mathcal{P}_0$; update R; set ρ
for each $(\mathbf{c}_i, \mathcal{P}_i) \in \mathcal{S} \setminus \{\mathbf{c}_0\}$ **do**
    compute p_i
$r \leftarrow p_0$; $m \leftarrow 1$
while $r < \tau_R$ **and** unscanned candidates remain **do**
    choose i with maximal p_i; scan $\mathcal{P}_i$; update $\mathcal{H}$
    $m \leftarrow m + 1$
    $\rho' \leftarrow$ distance to k-th in $\mathcal{H}$
    if $|\rho' - \rho| > \tau_\rho \rho$ **then**
        $\rho \leftarrow \rho'$; recompute p_j
    $r \leftarrow r = \sum_{i=0}^{m-1} p_i$
return R

Table 2: Mean single-threaded query latency and recall for APS variants on SIFT1M dataset at recall target 90%. **APS-RP:** recomputes probabilities after each partition scan without precomputation. **APS-R:** recomputes after each partition scan with precomputation. **APS:** recomputes probabilities only if query radius changes by more than $\tau_\rho = 1\%$, using precomputed beta function values.

Configuration	Recall	Search Latency
APS	91.2%	.48 ms
APS-R	91.2%	.59 ms
APS-RP	91.2%	.68 ms

during queries. Second, partition probabilities are recomputed only when the query radius ρ shrinks by more than a relative threshold τ_ρ. Table 2 shows these optimizations reduce query latency by 29% on SIFT1M without sacrificing recall.

6 Quake Implementation

Here we discuss NUMA-aware query processing and implementation details of Quake.

NUMA Data Placement and Query Processing Query processing in partitioned vector indexes is memory-bound, and therefore increasing the effective memory bandwidth available to the system will reduce query latency. NUMA-aware intra-query parallelism has been applied in the context of relational database systems to great success [17, 35], but has yet to be applied to vector databases.

In order to maximize memory bandwidth utilization, Quake distributes index partitions across NUMA nodes and ensures that cores only scan partitions resident in their respective node. Quake assigns index partitions to specific NUMA nodes using round-robin assignment. This assignment procedure allows for simple load balancing as partitions are added to the index by the maintenance procedure.

To maximize the benefits of data placement, Quake employs partition affinity and NUMA-aware work scheduling. Partitions are bound to specific CPU cores. This binding ensures that partitions are always scanned by the same core to maximize cache utilization. Queries are scheduled to worker threads based on the location of the data partitions they need to access. When a query requires scanning multiple partitions, the work is divided among threads on the relevant NUMA nodes where the partitions reside. By aligning thread execution with data placement, Quake minimizes remote memory accesses to maximize memory bandwidth utilization.

NUMA-Aware Query Execution with APS Quake integrates NUMA-aware processing with Adaptive Partition Selection (APS) to dynamically select which partitions to scan based on query requirements and desired recall. The query processing involves both worker threads scanning local partitions and a main thread coordinating the process.

Algorithm 2 NUMA-Aware Query Processing with Adaptive Partition Selection

Input: Query vector q, Index partitions P_i with locations $Node_j$, Recall threshold τ_R, Period to check recall T_{wait}
Output: Top-k nearest neighbors to q satisfying recall threshold τ

Initialize: $R \leftarrow \emptyset$ (global result set), $S \leftarrow$ sorted list of partitions based on distance to q (obtained from searching parent)
Distribute q to local memory of NUMA nodes
for all NUMA nodes $Node_j$ **in parallel do**
    $W_j \leftarrow$ worker threads on $Node_j$
    $P_j \leftarrow$ partitions on $Node_j$ from S
    Enqueue partitions P_j to local job queue
while not **all** partitions in S have been processed **do**
    Main Thread:
    Wait for a predefined interval T_{wait}
    Merge partial results from worker threads into R
    Estimate current recall r using Eqn. 7
    if $r \geq \tau_R$ **then**
        Break and terminate worker threads
Return top-k results from R
function WORKERTHREAD(q, Local Job Queue)
    while Job Queue not empty **do**
        $P_i \leftarrow$ **Dequeue** next partition from Job Queue
        Compute distances between q and vectors in P_i
        Update local partial results R_j
        Signal Main Thread about new partial results

Algorithm Explanation: In Algorithm 2, the main steps are:

1. **Initialization**: The query vector q is distributed to the local memory of NUMA nodes with relevant partitions. Partitions are sorted using their centroid distance to q.
2. **Worker Threads Execution**: Each NUMA node has worker threads that process partitions assigned to that node. They compute distances between q and vectors in their local partitions, updating their local partial results.
3. **Main Thread Coordination**: The main thread periodically merges partial results from all worker threads. It uses

the APS recall model to estimate the current recall based on the results accumulated so far.

4. **Adaptive Termination**: If the estimated recall meets or exceeds the threshold τ_R, the main thread returns the top-k results and signals the worker threads to terminate the processing of remaining partitions.

This adaptive approach ensures that the system processes only as much data as needed to meet the recall requirements, improving efficiency and reducing query latency.

Implementation Details We implemented Quake in 7,500 lines of C++ and provide a Python API for ease-of-use. We used primitives in Faiss [8], PyTorch [20], and SimSIMD [30] to enable high-performance management of inverted lists, efficient batch tensor operations, and AVX512 intrinsics for fast distance comparisons. We also used a high performance concurrent queue [29] to prevent contention during coordination of query processing. In addition, we developed a workload generator and evaluation framework in Python to create and evaluate vector search workloads. Quake is open-sourced at https://github.com/marius-team/quake.

7 Experiments

We evaluate Quake using a number of benchmarks and summarize our main findings:

1. Quake achieves the lowest search time across all dynamic workloads compared to state-of-the-art graph indexes, with 1.5-13× lower search latency than HNSW, DiskANN, and SVS while having 18-126× lower update latency.
2. We also find that APS matches the nprobe of an oracle across recall targets on SIFT1M, with only a 17-29% increase in latency relative to the oracle.
3. APS performs on-par or better than existing early termination methods [7, 18, 48] and requires no offline tuning.
4. Quake's NUMA-aware query processing exhibits linear scalability and high memory bandwidth utilization on the MSTURING100M dataset. Quake achieves 20× and 4× lower query latency compared to single-threaded and non-NUMA aware configurations, respectively.

7.1 Workloads

We performed our evaluation on a diverse set of real-world and synthetic workloads.

Wikipedia-12M This dataset and workload trace are derived from monthly Wikipedia page additions and page-view [4] frequencies between April 2013 and December 2021. We consider only pages about people or those linking to people. The dataset begins with 1.6 million pages and grows to 12 million after 103 updates, and therefore the average update size is ≈ 100,000 vectors. Embeddings are generated by training DistMult [46] graph embeddings (via Marius [28, 41]) on the Wikipedia link structure, and use the inner product metric.

The workload simulates monthly inserts of new pages, followed by 100,000 search queries sampling page embeddings with probability proportional to their page views corresponding to roughly a 50/50 read/write ratio. This setting imitates evolving interest and periodic growth of the dataset.

OpenImages-13M Using the methodology described by SVS [5], we generate a workload of 13M images from the Open Images dataset [15]. Embeddings are produced using Clip [37] in an inner product metric space. The workload maintains a sliding window of 2 million resident vectors and inserts and deletes vectors based on class labels until all 13 million vectors have been indexed at least once. Each insert and delete affects roughly 110K vectors. After each insert and delete operation, we run 1,000 queries randomly sampled from the entire vector set. This scenario stresses both insertion and deletion performance as well as sustained query latency.

Workload Generator To test performance under varying workload properties, we employ a configurable workload generator applicable to any vector dataset. The key parameters are: number of vectors per operation, operation count, operation mix (read/write ratio), and spatial skew. For skewed workloads, vectors are clustered and sampled from to produce queries and updates, reflecting hot spots in the vector space.

We construct two example workloads from a 10M vector subset of the MSTuring [2] dataset using L2 distance:

- **MSTuring-RO**: A pure search workload. We uniformly sample from 100,000 provided query vectors and execute 100 search operations, each querying 10,000 vectors. This setup tests search efficiency in a static setting.
- **MSTuring-IH**: A dynamic workload interleaving inserts and searches. Beginning with 1 million vectors, the dataset grows to 10 million as we process 1,000 operations with a 90% insert and 10% search ratio. This tests the ability to handle large-scale growth while maintaining query quality.

We use the datasets SIFT1M [13], SIFT10M, and MSTURING100M [2] to conduct microbenchmarks.

7.2 Experimental Setup

Large-scale experiments are run on a 4-socket server with Intel Xeon Gold 6148 CPUs (80 cores, 160 threads), 500 GB RAM across 4 NUMA nodes, and 300 GB/s total memory bandwidth. Some microbenchmarks (Tables 2, 5, 7, and 6) are run on a 2023 MacBook Pro with a M2 Max chip.

Search queries are processed one at a time and we report the total time to process all queries to reach a target of 90% recall for $k = 100$. Unless otherwise stated, all search numbers use a single worker thread. Quake additionally reports a multi-thread configuration **Quake-MT** (16 threads) where partition scans are parallelized for individual queries, while **Quake-ST** uses a single thread for search. For updates we report the total update and maintenance time, where both Quake and the baselines process updates in batches using 16 threads. This setup simulates an online environment where queries arrive individually, and updates are applied in batches. We report maintenance time separately from update latency, as maintenance can be conducted in the background in online

systems [45].

Baselines: We compare Quake against several state-of-the-art methods, including both partitioned and graph-based indexes:

- **Faiss-IVF** [8]: A popular inverted file (IVF) index in Faiss. It handles updates but does no maintenance.
- **DeDrift** [6]: An incremental maintenance strategy designed to reduce clustering drift by periodically reclustering large partitions together with small ones. We implement DeDrift's logic within Quake.
- **LIRE** [45]: Maintenance procedure used by SpFresh. LIRE incrementally splits large clusters and deletes small clusters after updates, followed by local reassignments. We implement LIRE's approach within Quake.
- **ScaNN** [10]: A state-of-the-art highly optimized partitioned index system. It uses an unpublished incremental maintenance procedure similar to LIRE.
- **Faiss-HNSW** [24]: A graph-based approach (HNSW) implemented in Faiss. It supports incremental inserts but not deletes. Thus, for workloads with deletions, we omit Faiss-HNSW from those comparisons.
- **DiskANN** [38]: System built around the Vamana [39] index with support for dynamic updates.
- **SVS** [5]: A recently released optimized implementation of the Vamana index with support for dynamic updates.

We configure the main parameters of Quake and the baselines as follows. We disable vector quantization/compression for all baselines, as not all baselines support it. For partitioned indexes we use $sqrt(|X_0|)$ partitions where $|X_0|$ is the initial number of vectors in the workload. For the graph indexes, we use a graph degree of 64. For LIRE and Quake, we set the partition refinement radius $r = 50$. For Quake we use a single level of partition, set $\tau = 250ns$, use one iteration of k-means for refinement and set f_M between 1%-10%. All systems use 16 threads for updates and maintenance (if applicable). SCANN, DiskANN, and SVS perform maintenance eagerly during an update, therefore we do not measure maintenance time separately from update time. We consider maintenance after each operation for all methods. Throughout all experiments, indexes search parameters are tuned to achieve an average of 90% recall for $k = 100$ across the workloads.

7.3 End-to-End Evaluation

Comparison with Baselines Table 3 shows that Quake consistently achieves lower search, update and total time on all workloads. On the WIKIPEDIA-12M workload, where the dataset grows over time and partitions can become unbalanced, the multi-threaded Quake-MT takes 1.53 hours to process searches, while single-threaded Quake-ST takes 9.48 hours. In contrast, Faiss-IVF climbs to 165 hours due to the lack of maintenance, DeDrift reaches 132 hours despite its rebalancing efforts, LIRE is unable to meet the recall target and takes 44 hours and SCANN performs similarly with poor update latency due to over-eager maintenance applied during updates. Even the best-performing graph-based method, DiskANN, takes 12 hours. Thus, Quake-MT is 8× faster to search than the strongest baseline on this workload derived from real-world access patterns.

On the OPENIMAGES-13M workload, which includes both insertions and deletions, Quake's multi-threaded and single-threaded search times are .03 and .14 hours respectively. The best competing approach, DiskANN, records .22 hours, making Quake-MT 7.3× and Quake-ST 1.6× faster. Faiss-HNSW does not support deletions so it is omitted. Both SVS's and DiskANN's delete consolidation is expensive, leading to orders of magnitude higher update latency than partitioned indexes, illustrating that graph-based indexes struggle with dynamic operations. Quake's continuous maintenance keep partitions balanced, achieving low latency and stable recall.

For the static, read-only, MSTURING10M-RO workload, Quake's maintenance improves the index structure even without data changes, adapting partitions to the query pattern. For Quake-MT, this yields a search time of .63 hours and Quake-ST takes 2.43 hours to conduct the search. However, the MSTURING10M dataset is especially challenging for partitioned indexes, as they need to scan roughly 10% of all partitions in order to meet the recall target. In contrast, the well-optimized SVS library exhibits a superior search time of .33 hours, demonstrating that in static settings, well-optimized graph indexes are strong competition.

On MSTURING10M-IH, where the dataset grows from one to ten million vectors, Quake-MT achieves a total search time of .54 hours. DiskANN, the second-best performer, has a search time of .81 hours, making Quake-MT 1.5× faster due to intra-query parallelism. However, single-threaded Quake is 2.6× slower than DiskANN, further illustrating the search efficiency of graph indexes. The other baselines fail to maintain the recall target or suffer from high latency due to their static parameters and inability to prevent partition skew.

Overall, these results demonstrate that Quake's combination of adaptive partition scanning, incremental maintenance, and NUMA-aware parallelism consistently delivers low-latency queries at the desired recall. Systems without maintenance (Faiss-IVF) suffer from skew-induced latency increases, those tied to static search parameters (LIRE) struggle to maintain recall without incurring higher query times, and graph-based methods (Faiss-HNSW, DiskANN) face substantial overheads when handling updates and deletions. By integrating these components, Quake matches the low update cost of partitioned indexes while outperforming graph indexes in search latency in dynamic workloads. Our design is an advancement to the state-of-the-art, providing stable, efficient performance across diverse, and evolving workloads.

Comparison with Partitioned Index Maintenance Methods Here we perform a detailed comparison with LIRE and DeDrift, measuring the latency, recall, and number of partitions over time on the WIKIPEDIA-12M workload. For a fair comparison, we use a single-thread to highlight the advantages of APS and maintenance in Quake. The results are

Method	WIKIPEDIA-12M				OPENIMAGES-13M				MSTURING10M-RO			MSTURING10M-IH			
	S	**U**	**M**	**T**	**S**	**U**	**M**	**T**	**S**	**M**	**T**	**S**	**U**	**M**	**T**
Quake-MT	1.53	.01	.44	**1.98**	.03	.02	.10	**.15**	.63	.08	.71	.54	.02	.14	**.70**
Quake-ST	9.48	.01	.44	9.93	.14	.02	.10	.26	2.43	.08	2.51	2.12	.02	.14	2.28
Faiss-IVF	165.8	.005	0	165.8^{+}	.45	.01	0	.46	12.25	0	12.25	13.72	.01	0	13.73
DeDrift	132.6	.03	.19	132.8^{+}	.23	.03	.19	.45	–	–	–	19.17	.03	.55	19.75
LIRE	44.2	.03	.38	44.61^{*+}	.15	.05	.11	.31	–	–	–	9.08	.02	.21	9.32*
ScaNN	50.27	1.75	0	52.02^{+}	.41	.21	0	.62	2.97	0	2.97	6.70	.09	0	6.79
Faiss-HNSW	14.65	.18	0	14.83	–	–	–	–	1.9	0	1.9	1.27	1.38	0	2.64
DiskANN	12.11	.32	0	12.43	.22	1.53	0	1.75*	1.16	0	1.16	.81	.48	0	1.28
SVS	20.54	.57	0	21.11*	.29	2.32	0	2.61	.33	0	**.33**	2.11	.24	0	2.35*

Table 3: Total workload time breakdown **in hours**. **S**: search, **U**: update, **M**: maintenance, **T**: overall total. Recall target = 90% and k=100. Search queries are processed one-at-time, updates are processed in batches, maintenance is conducted after each batch of search or update operations. *Denotes the method is unable to meet the recall target with static query parameters. $^{+}$Denotes the method did not finish in a 24 hour time budget, for these we estimate the runtime based on a 10% subsample of search queries.

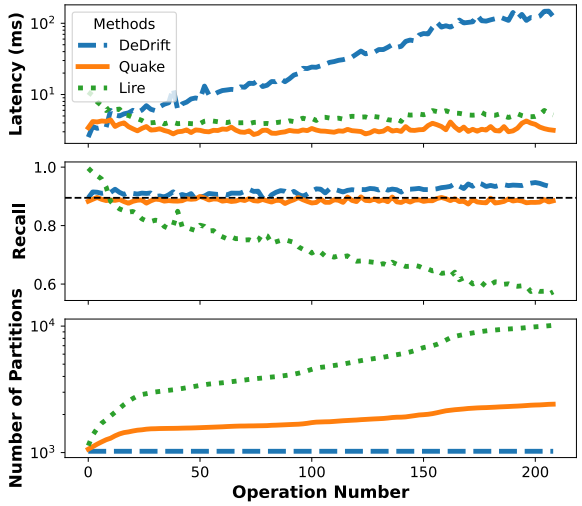


Figure 4: Comparison of single-threaded search latency, recall and number of partitions for Quake vs. maintenance approaches LIRE and DeDrift on WIKIPEDIA-12M. Quake maintains stable latency and recall throughout the workload.

shown in Figure 4. First looking at recall, we see that Quake maintains a stable recall of near 90%, while LIRE's recall degrades over time as it uses a static nprobe. DeDrift's recall stays relatively constant, as it does not adjust the number of partitions and therefore does not need to adjust nprobe. However, when turning our attention to latency, we see that Quake has near-constant stable latency, even as the dataset grows, while DeDrift's latency increases significantly with time. In terms of the number of partitions, we see DeDrift stays constant while Quake and Lire increase by 2.5× and 10× respectively. LIRE uses significantly more partitions because it uses size thresholding to determine when to split, regardless of whether a given partition is hot or not. Quake on the other hand only splits partitions if their contribution to the cost model is high, allowing for more efficient maintenance.

These results show that Quake's approach to maintenance is superior to existing methods for partitioned index maintenance in minimizing query latency and recall stability.

Table 4: Ablation Study on WIKIPEDIA-12M showing mean search latency and the standard deviation of recall.

Configuration	Search Latency	Recall Std.
Quake-MT	0.53 ms	.008
Quake-MT w/o APS	0.50 ms	.025
Quake-ST	3.28 ms	.005
Quake-ST w/o APS	3.18 ms	.025
Quake-ST w/o Maint/APS	45.20 ms	.014

Wikipedia-12M Ablation To quantify the contributions of Quake components, we disabled key features and measured the impact on WIKIPEDIA-12M workload in Table 4. We see that disabling APS has little impact on the query latency, as Quake can achieve a low latency even in the static nprobe setting. However, APS provides significantly more recall stability, as evidenced by the increase in standard deviation when APS is disabled. Disabling NUMA-aware multi-threading, however, shows a 6× increase in query latency, demonstrating the benefit of parallelization of partition scans. Finally, we disable maintenance and see a significant increase in latency, similar to the latency of Faiss-IVF; here partitions become extremely imbalanced due to the skew in the workload (see Figure 1a) causing queries to scan more vectors and therefore increasing latency. This further demonstrates the necessity for maintenance for dynamic workloads. In conclusion, each piece of Quake contributes to its performance in terms of both recall stability and minimal query latency.

7.4 Multi-Query Processing

Here we compare the search performance of Quake and baseline methods in a static batched query setting. Figure 5 shows the QPS at a recall of 90%, varying the number of queries in a batch on the WIKIPEDIA-12M workload. The dataset includes all 12M vectors, with 10,000 queries sampled accord-

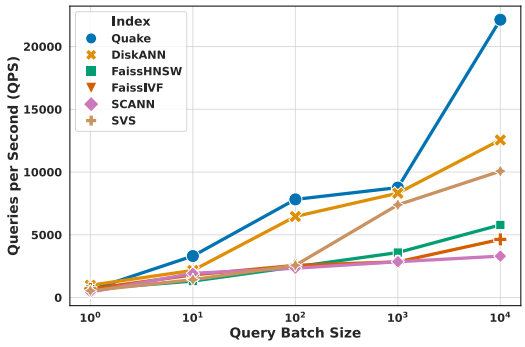


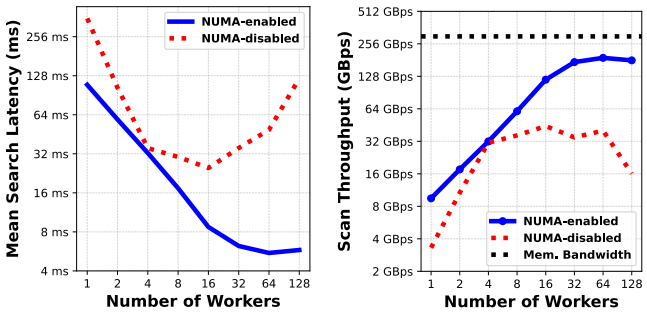
Figure 5: Multi-query evaluation on WIKIPEDIA-12M with 10,000 search queries. QPS @ recall=90% is measured for all baselines while varying the batch size. All methods use 16 threads to process queries.

ing to Wikipedia page views from December 2021. For Quake, FaissIVF, and SCANN, we use 3,500 partitions; for FaissHNSW, SVS, and DiskANN, we set the graph degree to 64. All experiments use 16 threads for query processing. Quake employs the multi-query execution policy in [26] and [34], grouping queries by the partitions they access and scanning each partition exactly once per batch in parallel.

Quake consistently outperforms all baselines across every batch size, with an increasing advantage as batch sizes grow. At the largest batch size (10,000 queries), Quake achieves a 6.7× speedup over FaissIVF and SCANN. This performance gain stems from Quake's efficient multi-query execution strategy, where it scans each partition once per batch, in contrast to FaissIVF and SCANN, which scan partitions individually per query. Compared to DiskANN, the strongest graph-based competitor, Quake still maintains a substantial 1.8× speedup. These results demonstrate that Quake delivers high performance not only in single-query scenarios but also in large multi-query workloads.

7.5 Scalability

We tested Quake's parallel scalability by varying the number of threads. In Figure 6 we measure the mean search latency and scan throughput (bytes scanned / query latency) on MSTURING100M to reach a recall of 90%. Note that this dataset has 100 million vectors and is 10× larger than the datasets we compared against previously. We compare our NUMA-aware parallelism with one in which NUMA is disabled. For both configurations, we see near linear scalability up to around 8 workers, where the non-NUMA latency performs best (28ms). The NUMA configuration however further improves and at 64 workers achieves a latency of 6ms. Looking at the scan throughput, we see that NUMA achieves a peak throughput of 200GBps. We do not completely saturate memory bandwidth due to other overheads involved in query processing (topk sorting, memory allocations, coordination). In conclusion, NUMA-aware intra-query parallelism is an effective mechanism for decreasing query latency by utilizing the full memory capabilities of multi-core machines.



(a) Mean Search Latency (b) Scan Throughput

Figure 6: MSTURING100M: Scaling the number of threads with and without NUMA.

Table 5: Early-termination methods on SIFT1M with a partitioned index with 1000 partitions. Each row shows the average recall, nprobe, and mean per-query latency in milliseconds over 10000 queries after tuning for a specific recall target for $k = 100$. We also report the total offline tuning time in seconds, where APS needs *no* offline tuning.

Method	Target	Recall	*nprobe*	Latency	Offline Tuning
APS	80%	82.1%	11.8	0.34ms	**0**
	90%	91.2%	20.2	0.48ms	**0**
	99%	98.9%	50.1	0.96ms	**0**
Auncel [48]	80%	85.7%	16.4	0.41ms	66.3s
	90%	98.1%	73.8	1.29ms	73.8s
	99%	99.7%	95.9	1.61ms	83.2s
SPANN [7]	80%	81.6%	11	0.31ms	173s
	90%	90.2%	19	0.43ms	183s
	99%	99.0%	70	1.07ms	259s
LAET [18]	80%	81.3%	10.5	0.29ms	81s
	90%	90.5%	18.2	0.42ms	104s
	99%	99.0%	58.3	1.03ms	232s
Fixed	80%	81.7%	11	0.33ms	318s
	90%	90.3%	19	0.44ms	330s
	99%	99.0%	65	1.16ms	424s
Oracle	80%	83.3%	11.5	0.29ms	320s
	90%	92.4%	19.3	0.41ms	331s
	99%	99.2%	42.0	0.74ms	368s

7.6 Comparison with Early Termination Methods

Table 5 compares early-termination methods on SIFT1M, highlighting the tradeoff between query latency, tuning time, and recall, we do not include ground truth generation time in the tuning time for the baselines. APS analytically estimates recall at query time, **eliminating offline tuning entirely**, while achieving latency within 30% of the oracle across all recall targets. Fixed selects a static *nprobe* per target via an

expensive offline binary search (up to 424s), and SPANN similarly performs a binary search and tunes a centroid-distance threshold; both closely match recall targets but incur higher latency at 90% and 99% recall. LAET trains a per-query prediction model, incurring moderate tuning overhead (81–232s), and matches recall targets with slightly higher latency compared to APS. Auncel is the most similar method to APS, as it aims to analytically estimate recall using partition intersection volumes, however its volume estimation requires calibration, and it is a conservative method, overshooting recall. We tune Auncel by binary searching a geometric parameter (a), overshooting recall significantly (up to 8.1 pp) and increasing latency by up to 169% compared to APS. Finally, the Oracle, which scans the minimal amount of partitions per-query, serves as a practical lower bound on achievable latency, though with prohibitively high tuning cost. The tuning overhead of the baselines demonstrates a significant burden for online scenarios where the queries, data, and index change. The tuning burden worsens at scale, where ground truth generation cost grows linearly with the data size, and running queries multiple times to conduct binary search takes longer. APS thus provides near-optimal performance without tuning overhead, matching or exceeding all baselines.

Table 6: SIFT10M: Recall and per-level search latency (ℓ_0, ℓ_1, total) for a single-level baseline (L_0: 40,000 partitions; L_1: 1 partition) and two-level index (L_0: 40,000 partitions; L_1: 500 partitions), where recall targets are varied at each level. The single-level baseline is the first row of each $\tau_r(0)$ block.

$\tau_r(0)$	$\tau_r(1)$	Recall	ℓ_0 (ms)	ℓ_1 (ms)	Total (ms)
80%	—	81.2%	2.07	4.85	6.92
	80%	74.8%	1.44	0.72	2.16
	90%	78.3%	1.56	1.19	2.75
	95%	80.1%	1.67	1.69	3.37
	99%	81.0%	1.75	2.57	4.33
	100%	81.1%	1.82	3.81	5.63
90%	—	91.3%	2.85	4.89	7.86
	80%	84.1%	2.07	0.77	2.84
	90%	88.2%	2.26	1.24	3.50
	95%	90.1%	2.38	1.72	4.10
	99%	91.0%	2.48	2.60	5.08
	100%	91.2%	2.62	3.88	6.50
99%	—	99.0%	4.82	5.5	10.3
	80%	91.4%	4.11	0.77	4.88
	90%	96.0%	4.59	1.27	5.86
	95%	97.7%	4.80	1.75	6.55
	99%	98.7%	5.08	2.65	7.74
	100%	98.9%	5.28	3.93	9.21

7.7 Multi-Level Recall Estimation

Here we evaluate the effectiveness of APS in a two-level partitioned index by measuring the impact of varying per-level recall targets on overall recall. Table 6 reports results on SIFT10M using 40,000 partitions at level L_0 and 500 partitions at level L_1. We use an initial search fraction of $f_M = 1.5\%$ at L_0 and 25% at L_1.

We observe that overly aggressive early termination at L_1 by setting $\tau_r(1)$ too low leads to a clear degradation in total recall. For instance, at $\tau_r(0) = 90\%$, reducing $\tau_r(1)$ from 99% to 80% lowers overall recall from 91.0% to 84.1%. This confirms that accurate recall estimation at the upper level is necessary to maintain accurate end-to-end recall estimates. These results justify our design decision in Section 5.1 to fix $\tau_r(1) = 99\%$ in multi-level configurations so that only the base recall target $\tau_r(0)$ needs to be provided.

In addition, the two-level index substantially reduces centroid scanning overhead. The single-level baseline must evaluate distances to all 40,000 centroids per query. In contrast, the two-level configuration performs an approximate search over the centroids. For example, at a recall target of $\tau_r(0) = 90\%$ with $\tau_r(1) = 99\%$ the total query latency drops from 7.86ms to 5.08ms, a 35% reduction, driven mainly by the drop in L_1 latency from 4.89ms to 2.60ms.

7.8 Maintenance Ablation

To understand the effectiveness of the primary components of adaptive incremental maintenance (cost-model, partition refinement, and rejection), we replay a dynamic SIFT1M trace (30% inserts, 20% deletes, 50% queries) with different components disabled. All methods use a single-thread and search using APS with k=100 and a 90% recall target. We also include LIRE as a baseline. Table 7 reports cumulative times in seconds. For configurations with refinement we use a refinement radius of $r_f = 50$. The full Quake policy delivers the lowest search cost (86s) while meeting the recall target. If we keep the cost model but skip refinement (NoRef), maintenance time decreases significantly from 21s to 5s, yet recall slips by 2.4 pp and the search time increase by 15.4s. This shows that while refinement is the dominant cost in maintenance, it is necessary for minimizing search latency. Disabling the cost model and instead using size-based thresholding (NoCost) shows why naive size thresholds are inadequate: search time rises 8% despite similar maintenance effort. The rejection mechanism is critical; once removed (NoRej), recall collapses to 66% even though search and maintenance appear cheap. LIRE, which relies solely on size thresholding, is 17% slower in search latency, confirming that the cost model, rejection mechanism, and partition refinement are essential for maintaining both index performance and quality.

8 Discussion

Here we offer a discussion of Quake's system parameters, how the design extends to new hardware and use-cases.

8.1 Setting System Parameters

Quake exposes a few search and maintenance parameters, which we fix across all workloads unless otherwise stated. These defaults give stable performance with minimal tuning.

Table 7: Maintenance ablation on the SIFT1M workload. Times are cumulative (in seconds) over the course of the workload. Recall is averaged over all queries.

Maintenance Variant	Search	Update	Maint.	Recall
Quake (Full)	86.3s	21.7s	21.4s	90.5%
NoRef	101.7s	22.2s	5.2s	88.1%
NoRef+NoRej	85.5s	21.0s	1.0s	73.0%
NoRej	84.2s	19.6s	18.5s	66.2%
NoCost	93.5s	20.0s	20.4s	90.1%
NoCost+NoRef	100.7s	21.2s	0.8s	87.9%
LIRE	100.5s	21.2s	11.9s	90.0%

Search parameters. The *initial candidate fraction* f_M determines the number of partitions to consider in APS. It has the largest impact on performance. If set too low, APS may not meet the recall target; if too high, the initial scan dominates latency. We set this between 1% and 10%. In future work, we aim to remove this parameter entirely. The *number of worker threads* is best set to the number of physical cores. For large problem sizes, Quake scales linearly with thread count until memory bandwidth is saturated (Figure 6). The *recompute threshold* controls how often APS updates its recall estimate. We set this to 1%, which avoids unnecessary recomputation with negligible impact on recall (Table 2).

Maintenance parameters. The *split/merge threshold* τ sets the minimum predicted latency improvement required to trigger a split or delete. We set $\tau = 250$ns. Lower values increase maintenance cost and may cause over-splitting; higher values can allow imbalance to persist. The *split access scaling parameter* α estimates the drop in access frequency after a split. We fix $\alpha = 0.9$, which worked well across all benchmarks. If maintenance tuning is needed, we recommend keeping α fixed and adjusting τ. *Refinement* is controlled by two parameters: the refinement radius r_f (number of nearby partitions considered) and the number of refinement iterations. We use one iteration over the 50 nearest partitions. From our ablation study, disabling refinement reduces maintenance time by 75% but increases query latency and reduces recall (Table 7). The *window size* for access frequency statistics is set equal to the maintenance interval. For example, if maintenance runs every 100,000 queries, the window also spans 100,000 queries. Smaller windows adapt faster but are more volatile.

Most parameters are fixed across workloads. In practice, only the initial candidate fraction f_M and the maintenance threshold τ benefit from tuning; however, their defaults are sufficient for the workloads we evaluated.

8.2 Deployment Considerations

Quake's design can be extended to support a range of hardware and use-cases through minor changes to the cost model and APS logic. We reserve these extensions for future work.

Heterogeneous Hardware To adapt to devices with varying scan throughput such as CPUs, GPUs, or disk-backed storage, the cost model can be modified by profiling per-device scan latency (e.g., $\lambda_{\text{CPU}}(s)$, $\lambda_{\text{GPU}}(s)$, $\lambda_{\text{Disk}}(s)$) and updating partition cost estimates accordingly.

Distributed Environments In a distributed setting, each machine can run APS and maintenance independently on its local partitions. The cost model can account for partition locality and inter-node variation. A separate load balancer would be required to assign and replicate partitions across machines in a way that minimizes total query cost.

Vector Compression Vector compression techniques, such as Product Quantization [14], reduce scan cost. Quake can support compression by profiling scan latency over partitions of compressed vectors and updating $\lambda(s)$ in the cost model.

Filters Filtered queries can be supported by scaling per-partition recall probabilities in APS, based on the estimated number of items that pass the filter in each partition. This will enable Quake to avoid scanning partitions unlikely to contain matching results while preserving recall targets.

Concurrency The current implementation executes searches, updates, and maintenance serially. Quake can support concurrency through copy-on-write semantics, allowing background operations to build new index views while readers continue on the current one without blocking.

9 Conclusion

Experimental results show that Quake reduces query latency compared to baseline approaches under dynamic and skewed workloads, without requiring manual tuning. It achieves high recall, matching the performance of an oracle for setting the query parameter nprobe. Compared to existing partitioned indexes like Faiss and SCANN, Quake reduces query latency by A) adaptively maintaining index partitions and B) maximizing memory bandwidth during query processing. Compared to graph indexes like SVS, HNSW, and DiskANN, Quake offers more efficient indexing and updates while matching or reducing query latency. In summary, our evaluation shows Quake minimizes query latency while meeting recall targets on dynamic workloads with skewed access patterns.

Acknowledgments We would like to thank our shepherd, Nitin Agrawal, and the reviewers for their valuable feedback and efforts in making this a stronger paper. This work was supported by NSF grant CNS-2237306, Apple Scholars in AIML PhD Fellowship, and UW-Madison Hilldale Undergraduate Research Fellowship. This work was also supported by DARPA under the grant AIE DARPA-PA-22-01. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Qdrant - Vector Database. https://qdrant.tech/.

[2] Billion-scale approximate nearest neighbor search challenge: Neurips'21 competition track. https://big-ann-benchmarks.com/, 2021.

[3] Vector database for vector search | pinecone. https://www.pinecone.io, 2024. Accessed on December 4, 2023.

[4] Wikipedia:pageview statistics. https://en.wikipedia.org/wiki/Wikipedia:Pageview_statistics, 2024.

[5] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore Willke, and Mariano Tepper. Locally-adaptive quantization for streaming vector search. *arXiv preprint arXiv:2402.02044*, 2024.

[6] Dmitry Baranchuk, Matthijs Douze, Yash Upadhyay, and I. Zeki Yalniz. DeDrift: Robust Similarity Search under Content Drift, August 2023. arXiv:2308.02752 [cs].

[7] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search.

[8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2024.

[9] Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 311–320, 2018.

[10] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning*, pages 3887–3896. PMLR, November 2020. ISSN: 2640-3498.

[11] Malay Haldar, Mustafa Abdool, Prashant Ramanathan, Tao Xu, Shulin Yang, Huizhong Duan, Qing Zhang, Nick Barrow-Williams, Bradley C Turnbull, Brendan M Collins, et al. Applying deep learning to airbnb search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1927–1935, 2019.

[12] Helia Hashemi, Aasish Pappu, Mi Tian, Praveen Chandar, Mounia Lalmas, and Benjamin Carterette. Neural instant search for music and podcast. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2984–2992, 2021.

[13] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[14] Herve Jégou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, January 2011. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[15] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *International Journal of Computer Vision*, 128(7):1956–1981, March 2020.

[16] Yongjae Lee and Woo Chang Kim. Concise formulas for the surface area of the intersection of two hyperspherical caps. *KAIST Technical Report*, 2014.

[17] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 743–754, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Conglong Li, Minjia Zhang, David G Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2539–2554, 2020.

[19] Shengqiao Li. Concise formulas for the area and volume of a hyperspherical cap. *Asian Journal of Mathematics & Statistics*, 4(1):66–70, 2010.

[20] LibTorch: PyTorch C++ API. https://pytorch.org/cppdocs.

[21] David C Liu, Stephanie Rogers, Raymond Shiau, Dmitry Kislyuk, Kevin C Ma, Zhigang Zhong, Jenny Liu, and Yushi Jing. Related pins at pinterest: The evolution of a real-world recommender system. In *Proceedings of*

the 26th international conference on world wide web companion, pages 583–592, 2017.

[22] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. Monolith: Real time recommendation system with collisionless embedding table. In *5th Workshop on Online Recommender Systems and User Modeling (ORSUM2022), in conjunction with the 16th ACM Conference on Recommender Systems*, 2022.

[23] Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. Cracking vector search indexes. *arXiv preprint arXiv:2503.01823*, 2025.

[24] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, April 2020.

[25] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Umar Farooq Minhas, Jeffery Pound, Cedric Renggli, Nima Reyhani, Ihab F Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. Incremental ivf index maintenance for streaming vector search. *arXiv preprint arXiv:2411.00970*, 2024.

[26] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. High-Throughput Vector Similarity Search in Knowledge Graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–25, June 2023.

[27] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. Quake: Adaptive indexing for vector search (technical report). *arXiv preprint arXiv:2506.03437*, 2025.

[28] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 533–549, 2021.

[29] moodycamel::ConcurrentQueue. https://github.com/cameron314/concurrentqueue.

[30] SimSIMD. https://github.com/ashvardanian/SimSIMD.

[31] Jiongkang Ni, Xiaoliang Xu, Yuxiang Wang, Can Li, Jiajie Yao, Shihai Xiao, and Xuecang Zhang. DiskANN++: Efficient Page-based Search over Isomorphic Mapped Graph Index using Query-sensitivity Entry Vertex, November 2023. arXiv:2310.00402 [cs].

[32] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. Embedding-based news recommendation for millions of users. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1933–1942, 2017.

[33] Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. Pinnersage: Multi-modal user embedding framework for recommendations at pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2311–2320, 2020.

[34] Jeffrey Pound, Floris Chabert, Arjun Bhushan, Ankur Goswami, Anil Pacaci, and Shihabur Rahman Chowdhury. Micronn: An on-device disk-resident updatable vector database. *arXiv preprint arXiv:2504.05573*, 2025.

[35] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.*, 10(2):37–48, October 2016.

[36] An Qin, Mengbai Xiao, Yongwei Wu, Xinjie Huang, and Xiaodong Zhang. Mixer: efficiently understanding and retrieving visual content at web-scale. *Proceedings of the VLDB Endowment*, 14(12):2906–2917, 2021.

[37] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[38] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[39] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[40] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. Soar: Improved indexing for approximate nearest neighbor search. In *Neural Information Processing Systems*, 2023.

[41] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2023.

[42] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[43] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 839–848, 2018.

[44] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, 2020.

[45] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 545–561, New York, NY, USA, October 2023. Association for Computing Machinery.

[46] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*, 2014.

[47] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. {VBASE}: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 377–395, 2023.

[48] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. Fast, approximate vector queries on very large unstructured datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 995–1011, 2023.

A Artifact Appendix

Abstract

This artifact provides the experimental setup for Quake to reproduce key results. See the artifact's README for full details.

Scope

This artifact enables validation of Quake's core experimental findings, such as those related to Adaptive Partition Scanning (APS), NUMA-aware searching, and maintenance policies. Refer to the artifact's README for specific experiments.

Contents

The artifact includes Python scripts (e.g., `experiment_runner.py`, individual experiment `run.py` files), system installation scripts (`install.sh`), experiment configurations (`configs/`), Conda environment files, and the paper PDF. The README details the full directory structure and contents.

Hosting

The artifact is part of the Quake repository in the `osdi2025` branch. located at https://github.com/marius-team/quake/tree/osdi2025/test/experiments/osdi2025. Use the latest commit on this branch.

Requirements

Python 3.9+ and Conda are required. Tested on Linux (Ubuntu 22.04). Some experiments (e.g., NUMA evaluations) require a machine with NUMA for meaningful reproduction. The `install.sh` script lists system-level dependencies. Refer to the README for comprehensive requirements.

A.1 Installation

Follow the detailed installation instructions in the artifact's README. Options include a comprehensive setup using `install.sh` or a Quake-only Conda environment setup.

A.2 Experiment Workflow

All experiments are launched via `experiment_runner.py` from the repository root. Experiments will typically download and prepare required datasets if not found locally. Experiments are run as follows:

```
python3 -m test.experiments.osdi2025.experiment_runner
-experiment kick_the_tires - config sift1m
```

After execution, experiments print status updates and save results (e.g., CSV files, plots) to an output directory, as indicated in the console output. The artifact's README provides the complete command structure, detailed explanations of all parameters (including `-output-dir`), further examples, and a full summary of available experiments with their specific configurations.



Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD

Hao Guo Youyou Lu*

Tsinghua University

Abstract

We propose PipeANN, an on-disk graph-based approximate nearest neighbor search (ANNS) system, which significantly bridges the latency gap with in-memory ones. We achieve this by aligning the best-first search algorithm with SSD characteristics, avoiding strict compute-I/O order across search steps. Experiments show that PipeANN has 1.14×–2.02× search latency compared to in-memory Vamana, and 35.0% of the latency of on-disk DiskANN in billion-scale datasets, without sacrificing search accuracy.

1 Introduction

High-dimensional vectors with tens or hundreds of dimensions are powerful data representations [17, 19]. Vector search, which searches a dataset for the closest neighbors given a target vector, is used in various scenarios such as recommendation [14, 15, 27] and retrieval augmented generation (RAG) [12]. It is not efficient to do accurate vector searches for high-dimensional vectors [9], so *approximate nearest neighbor search* (ANNS) is preferred, which returns an approximated set of k nearest neighbors (i.e., top-k). Among all types of ANNS indexes, graph-based indexes [6, 16], where vectors are organized as a directed graph, are favored for their low search latency under high accuracy.

To support large-scale datasets (e.g., billions of vectors [6, 14, 21, 27]), an increasing number of organizations [4, 21, 33] prefer solid-state drives (SSDs) for storing ANNS indexes, due to their cost-efficiency. However, graph-based indexes fail to maintain low search latency on disk as in memory. As shown in Figure 1, the on-disk DiskANN has significantly higher search latency than the in-memory Vamana (e.g., 4.18× for 0.9 recall and 3.14× for 0.99 recall).

By analysis, we find the high latency is caused by the intrinsic mismatch between graph-based ANNS algorithms and the I/O characteristics of SSDs, namely *long I/O latency* and *asynchronous, parallel I/O*: Graph-based ANNS follow the

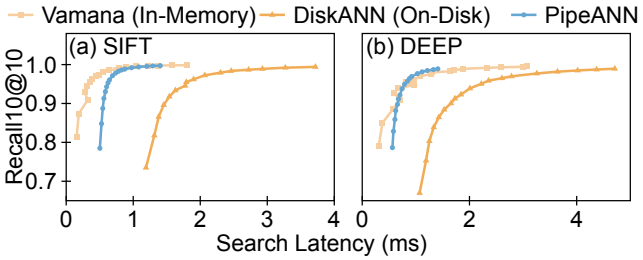


Figure 1: Latency gap between in-memory (Vamana [23]) and on-disk (DiskANN [23]) graph-based indexes in two datasets. Our system (PipeANN) significantly bridges the gap.

best-first search algorithm to explore vectors in the graph — in each search step, it explores the best neighbors (i.e., the nearest ones) of all the explored vectors, to reduce the number of accessed vectors per search. However, two issues prevent it from achieving low latency on disk. First, the best-first algorithm incurs *ordered compute and I/O* across search steps. It harms search latency because of the long (e.g., 7.43× than compute latency, §2.2) but wasted I/O latency which fails to overlap with compute. Second, the best-first algorithm forces *synchronous I/O* in each search step, where it batch-reads the nearest neighbors synchronously. It induces an underutilized I/O pipeline (e.g., 76% utilized, §2.2) when waiting for slow reads in the batch.

In this paper, **we seek to align the search algorithm with SSD I/O characteristics**. We find this idea feasible without affecting search convergence, by observing that the best-first algorithm is not a must: Unlike scalar indexes (e.g., B$^+$-tree) where objects have one search path, graph-based ANNS indexes have multiple search paths for each vector, considering its multiple in-edges. The best-first algorithm only estimates one short search path, not the unique one. Thus, a tweaked search algorithm can exploit other paths for convergence.

We propose PipeSearch, an on-disk graph-based ANNS algorithm that achieves low latency by aligning the best-first search algorithm with SSD I/O characteristics. The key ob-

*Youyou Lu is the corresponding author (luyouyou@tsinghua.edu.cn).

servation for such alignment is *the pseudo-dependency of compute and I/O in best-first search*: In each search step, the neighbors to be read can be decided by only the in-memory *candidate pool* containing the neighbor IDs, without waiting for ongoing I/O or compute (i.e., neighbor exploration) to finish. Thus, PipeSearch avoids strict compute-I/O order across search steps: When the I/O pipeline is not full, PipeSearch fills it by asynchronously reading the current nearest neighbors in the candidate pool, regardless of ongoing I/O or unfinished compute. Overlapped with I/O, PipeSearch explores the pre-read neighbors in a best-effort manner.

PipeSearch brings significant performance benefits. On the one hand, compute and I/O are overlapped, which accelerates ANNS as they show close (the same order of magnitude) latency. On the other hand, the I/O pipeline is better utilized. Experiments demonstrate the low latency of PipeSearch, which has ~50% latency compared to best-first search.

PipeSearch achieves low latency but at the cost of throughput. There are two challenges to increasing its throughput. *The first* is how to dynamically adjust to suitable pipeline widths during a single search, in order to leverage the high throughput of narrow pipelines and the low latency of wide ones simultaneously. *The second* is how to avoid the accumulation of read-but-unexplored neighbor vectors caused by wide pipeline and slow neighbor exploration, which induces sub-optimal I/O decisions and reduced search throughput.

To this end, we implement PipeANN, a low-latency ANNS system with high search throughput. PipeANN tackles the two challenges in PipeSearch with two techniques:

First, we dynamically increase the I/O pipeline width during the search, instead of keeping a static pipeline width, to benefit from high throughput and low latency simultaneously. The throughput drop in large pipeline widths mainly arises from *I/O waste*, which makes it easier to saturate SSD IOPS because of more I/O per search. However, we find that I/O waste decreases as the search progresses. This is due to the growing number of unexplored top-k neighbors during the search. In later search steps, the candidate pool contains more unexplored top-k neighbors, which allows a wider I/O pipeline with little I/O waste.

Second, we ensure an upper bound for the number of missed neighbors (i.e., ongoing I/O and read-but-unexplored neighbors) when deciding on each I/O, instead of ensuring a full I/O pipeline at all times, to avoid neighbor accumulation. Specifically, when multiple I/Os finish simultaneously, we repeatedly explore one neighbor and issue one I/O, instead of issuing multiple I/Os to fill up the pipeline. Thus, we strike a balance between a full I/O pipeline (PipeSearch) and reduced I/O waste (best-first search), increasing throughput while sacrificing little latency.

We evaluate PipeANN to show its efficiency in low-latency ANNS. As shown in Figure 1, PipeANN significantly bridges the latency gap between in-memory and on-disk ANNS, especially for high recall (≥ 0.9). Compared with state-of-the-art ANNS indexes on disk, PipeANN shows at least 70.6% lower latency and 1.35× higher throughput for 0.9 recall. In billion-scale datasets, PipeANN has 35.0% latency and 1.71× higher throughput compared to DiskANN [23].

While primarily designed for SSDs, PipeANN's techniques could also be adopted to other storage media, such as remote memory. These storage media exhibit similar I/O characteristics as SSDs, namely μs-scale I/O latency and parallel I/O, so ANNS indexes stored on these media [5] could improve their performance by adopting PipeANN.

PipeANN has its limitations. Despite the techniques for throughput, PipeANN's speculative I/O still leads to lower search throughput than the greedy I/O in best-first search. However, we believe PipeANN's latency-throughput tradeoff is worthwhile in applications with ms-scale latency budgets (e.g., large-scale search and recommendation [6, 21]), where PipeANN could benefit from lower search latency or higher search accuracy with the same latency.

In summary, this paper makes the following contributions:

- We find the best-first search algorithm restricts the design space in exploiting SSD characteristics, inducing high latency for on-disk graph-based ANNS (§2).
- We design PipeSearch, a low-latency graph-based ANNS algorithm on disk by aligning the best-first search algorithm with SSD (§3).
- We implement PipeANN, a low-latency ANNS system with high search throughput. PipeANN increases the throughput of PipeSearch by dynamic pipeline and algorithm optimization (§4).
- We evaluate PipeANN to demonstrate its efficacy in achieving low-latency and high-throughput ANNS on disk (§5).

2 Background and Motivation

In this section, we first provide a primer for on-disk graph-based ANNS. Then, we characterize the best-first search algorithm to show its mismatch with SSD characteristics.

2.1 Graph-Based ANNS and Best-First Search

The k-nearest neighbor search problem aims to find the k nearest vectors of the target vector in a dataset. However, accurate search is challenging in high-dimensional vector spaces (e.g., hundreds of dimensions), which is known as the curse of dimensionality [9]. To tackle this challenge, approximate nearest neighbor search (ANNS) algorithms are proposed. Given the target vector, ANNS finds an approximate set of its top-k nearest vectors. Of all the ANNS algorithms, graph-based ones [6, 16, 23] show promising performance and accuracy. They organize the vectors as a directed graph, and vector search is conducted by graph traversal.

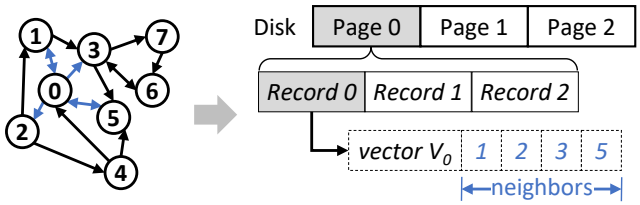


Figure 2: Index layout of on-disk graph-based ANNS.

Algorithm 1 Best-first search

```
1:  G ← graph, q ← query vector, W ← I/O pipeline width
2:  procedure BestFirstSearch(G, q, W)
3:      s ← starting vector, L ← candidate pool length
4:      candidate pool P ← {<s>}, explored pool E ← ∅
5:      while P ⊆ E do
6:          V ← top-W nearest vectors to q in P, not in E
7:          Read V from memory or disk
8:          E.insert(V)
9:          for nbr in V.neighbors do
10:             P.insert(<nbr, Distance(nbr, q)>)
11:         end for
12:         P ← l nearest vectors to q in P
13:     end while
14:     return k nearest vectors to q in E
15: end procedure
```

Graph layout. To support billion-scale vector search [6,21], existing systems [23,25] store the graph index on disk and do on-disk graph-based ANNS. As shown in Figure 2, the graph is stored on disk as multiple *records*. Each record consists of a vector and all its neighbor IDs.

Best-first search algorithm. Existing graph-based vector search systems, either in memory [6, 16] or on disk [23,25], take a best-first search algorithm for graph traversal [1,26]

As shown in Algorithm 1, best-first search maintains a candidate pool with a fixed *candidate pool length* (denoted as L in this paper), containing current top-L nearest vectors. The search starts with a fixed starting vector and contains multiple search steps. Each step explores the top-W nearest unexplored vectors in the candidate pool, by reading them in a batch and inserting their neighbors into the candidate pool. The search terminates when all the vectors in the candidate pool are explored. We use W to represent *I/O pipeline width* (i.e., the maximum number of parallel I/O requests) in this paper. This definition equals the beam width in best-first search [23].

For in-memory indexes [6, 16], reading the vectors (line 7) takes a short time, so they use $W = 1$ to reduce the waste of compute and I/O. However, SSD has high I/O latency, so it takes more time to read a vector than explore it for on-disk indexes [23, 25]. Therefore, they use $W > 1$ to utilize the SSD I/O pipeline. Also, they use PQ-compressed vectors in memory to calculate the distance with neighbors (line 10), without disk reads. In each search step, only W disk reads are conducted. In this paper, best-first searches with $W = 1$ and $W > 1$ are also called *greedy search* and *beam search*.

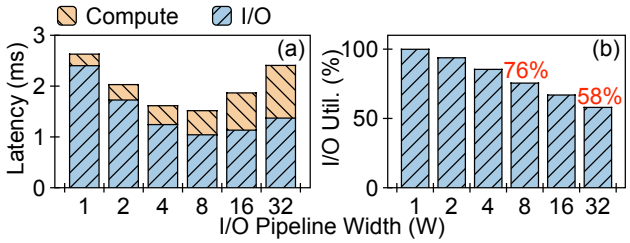


Figure 3: (a) Search latency breakdown. (b) I/O pipeline utilization rate (average #ongoing I/O ÷ I/O pipeline width).

Low-latency ANNS. Achieving low latency ANNS is beneficial for real-world applications. First, real-world systems have strict latency demands. In large-scale search or recommendation systems [6,21], billion-scale, or even larger, vector searches should complete within ~10ms to meet response time requirements. Second, for ANNS systems, a longer search time allows exploring more vectors, thus leading to higher search accuracy. Therefore, if we could reduce the search latency at the same accuracy, we could simultaneously increase search accuracy within the same latency demands.

2.2 Best-First Search Mismatches with SSD

Graph-based ANNS shows low latency in memory [6, 16], but fails to maintain such latency on disk. We conduct experiments to demonstrate this issue. We do best-first searches with different Ws, using a graph index built on 100 million vectors in the SIFT dataset [10]. The target recall is 90%, as recommended by the BigANN benchmark [20].

As shown in Figure 1(a), the search latency on disk is 4.18× compared to that in memory. By analysis, we find the cause of long latency is that *best-first search algorithm mismatches with SSD I/O characteristics*. Specifically, we focus on the following two SSD I/O characteristics:

- **Long I/O latency.** Microsecond or tens of microseconds latencies, common to flash-based SSDs.
- **Asynchronous, parallel I/O.** The ability to handle multiple (e.g., 32) in-flight read requests in parallel.

We find two issues as follows.

Issue 1: Ordered compute and I/O across search steps. The best-first search algorithm naturally causes data dependency, inducing ordered compute and I/O. In each search step, the best-first search batch reads and explores the W nearest unexplored neighbors to the target vector. Therefore, the current I/O batch depends on the previous search step, namely the previous compute and the previous I/O batch.

Such ordered compute and I/O show little overhead in memory but harm the ANNS latency on disk. This is because, when

conducting ANNS in memory, the memory access latency is orders of magnitude lower than compute, which is typically executed serially in a search thread. When on disk, I/O latency is higher than compute, which shifts the bottleneck to I/O. Figure 3(a) demonstrates this issue by breaking down the latency. When $W = 1$ (i.e., greedy search), compute latency is only 9.5% of I/O latency. Even when using $W = 8$ (the lowest overall latency) to utilize I/O parallelism, compute latency is still 45.6% of I/O latency. Unfortunately, the long I/O latency is wasted, as it fails to overlap with compute.

Issue 2: Synchronous I/O in each search step. Best-first search uses $W > 1$ to batch-read multiple records in each search step, to utilize I/O parallelism. However, it still requires synchronously waiting for all the I/Os to finish.

Such synchronous I/O causes an *under-utilized I/O pipeline*. Figure 3(b) shows the I/O pipeline utilization rate (the average number of ongoing I/Os during I/O time ÷ W). During I/O time, the pipeline is only 76% full for $W = 8$, and 58% full for $W = 32$. This is because the I/O latency fluctuates in SSDs. Best-first search waits for a whole I/O batch to finish, so the latency depends on the tail latency of the I/Os. When waiting for some slow I/Os, the I/O pipeline is not fully utilized.

3 PipeSearch

Based on the observations above, we propose PipeSearch, a low-latency ANNS algorithm by aligning the best-first search algorithm with SSD hardware. In this section, we first show that this idea is feasible. Then, we introduce PipeSearch and analyze its performance benefits. Finally, we evaluate it to demonstrate its dilemma in latency and throughput.

3.1 Tweaking Best-First Search is Possible

One may think that tweaking the best-first search algorithm may prohibit the search from convergence. However, we argue that **the best-first algorithm is not a must and can be tweaked without affecting search convergence.**

This is enabled by multiple search paths in graph-based indexes: Unlike scalar indexes (e.g., B^+-tree) where each object has only one search path, in graph-based ANNS indexes, each vector can be found in multiple paths using its multiple in-edges. The best-first search only estimates a short search path in the graph, but not the unique path. Therefore, tweaking the search algorithm is possible. Although it may lengthen the search path, it does not prohibit search from convergence and brings more opportunities for reducing latency.

3.2 PipeSearch Algorithm

We propose PipeSearch, a low-latency algorithm for graph-based ANNS on disk. The key idea of PipeSearch is to align the best-first search algorithm with SSD characteristics. As shown in Figure 4, best-first search executes compute and

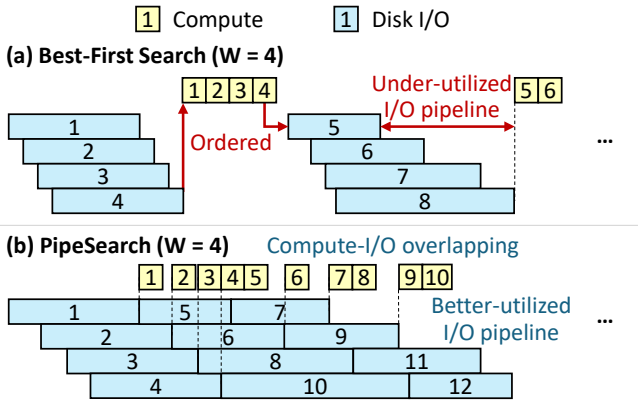


Figure 4: Comparison of PipeSearch with best-first search.

I/O in order, unfriendly to SSDs. In contrast, PipeSearch tweaks the algorithm to avoid such strict order, thus achieving compute-I/O overlapping and a better-utilized I/O pipeline.

Key observation: pseudo-dependency of compute and I/O. In each search step, the best-first search issues I/O and explores the nearest neighbors in order. However, such an order is not necessary: I/O can be decided only by the in-memory candidate pool, regardless of ongoing I/O and unexplored neighbors. Thus, when there are ongoing I/O, we can directly issue I/O for the nearest unread neighbor in the candidate pool, there is no need to wait for all the ongoing I/O to finish (like best-first search). Neighbor exploration can be executed in a best-effort manner, independent from disk I/O.

Algorithm overview. PipeSearch avoids strict compute-I/O order across search steps. Specifically, it maintains a candidate pool with a fixed length L, similar to best-first search. Also, it maintains an I/O pipeline Q with a specific width W, containing ongoing I/O. The search iteratively executes the following steps: If the I/O pipeline is not full, PipeSearch issues I/O to fill up the pipeline based on the current candidate pool. Overlapped with I/O, it explores the nearest vector in an unexplored set U and updates the candidate pool using its neighbors. Then, it polls for I/O completion and adds all the vectors acquired to the unexplored set U.

3.3 PipeSearch Reduces Search Latency

PipeSearch achieves low-latency graph-based ANNS on disk by tackling the two issues in §2.2. We analyze it as follows.

PipeSearch achieves compute-I/O overlapping, which accelerates ANNS as they show close latency. Both short and long I/O latencies may make pipelining inefficient. If I/O latency is short like in memory, greedy search shows low latency. If I/O latency is long, pipelining will degrade to best-first search due to short compute. However, in graph-based ANNS on disk, compute and I/O latencies are of the same order of magnitude, making PipeSearch efficient. As

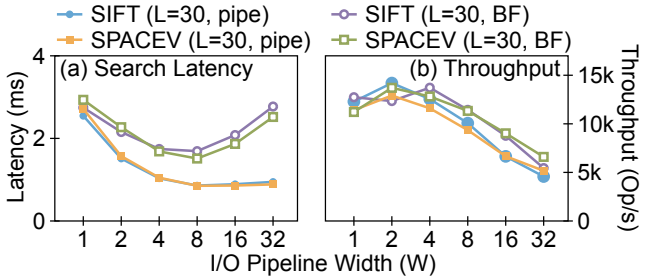


Figure 5: Latency and throughput of PipeSearch and best-first search (BF) with different Ws. With $L = 30$, both algorithms achieve 90% accuracy in terms of recall10@10.

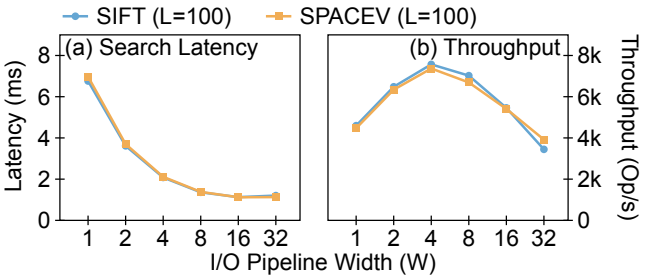


Figure 6: Latency and throughput of PipeSearch with different Ws and $L = 100$. PipeSearch achieves 99%/97% accuracy in terms of recall10@10 in SIFT/SPACEV.

shown in Figure 3(a), when $W = 32$, the compute latency is 75.6%/72.7% of the I/O latency. Overlapping them is possible to provide a 1.7× performance boost.

The I/O pipeline can be saturated by enough I/O, because of the navigation graph feature. In a typical navigation graph, each vector contains hundreds of neighbors, so the candidate pool contains hundreds of unexplored vectors after the first search step, which can be used to fill up the pipeline. It is possible to provide another 1.7× performance boost when $W = 32$, as in Figure 3(b). There may not be enough vectors to fill up the pipeline in the later search steps when most vectors are recalled. However, in §4.2, we find it does not last long.

3.4 Dilemma of Latency and Throughput

PipeSearch reduces the latency of best-first search but fails to achieve low latency and high throughput simultaneously. We evaluate PipeSearch using 100 million vectors in two datasets, SIFT [10] and SPACEV [4], using 1 thread for latency and 56 threads for throughput. As a comparison, we evaluate the best-first search with the same W. The results are shown in Figure 5, and we make the following observations:

(1) PipeSearch fails to ensure low latency and high throughput simultaneously, with a static pipeline width. On the one hand, the pipeline width with the lowest search latency (i.e., $W = 8$) fails to achieve high throughput simultaneously. It has 71.0%/72.4% lower throughput compared to $W = 2$. On the other hand, the optimal pipeline width varies in different configurations. Figure 6 shows the results with $L = 100$, where $W = 16$ shows the lowest latency (82.7%/81.1% compared to $W = 8$), and $W = 4$ delivers the highest throughput.

(2) Compared to best-first search, PipeSearch reduces latency but degrades throughput. When $W = 8$, PipeSearch has 50.7%/56.3% lower latency than best-first search in the SIFT/SPACEV dataset. However, the throughput degrades to 88.1%/82.5%. This issue is more significant for larger pipeline widths. When $W = 16$, PipeSearch only achieves 75.8%/73.8% of the throughput of best-first search.

The throughput degradation results from I/O waste. I/O waste, which increases average I/O per search, makes PipeSearch easier to saturate SSD bandwidth and thus causes a throughput drop. It arises from two aspects. The first is large pipeline widths. For PipeSearch, $W = 32$ has 2.44×/2.24× average I/O per search compared to $W = 8$, due to more speculative I/O. Although a small pipeline width reduces I/O waste, it leads to more sequential accesses, thus increasing the number of search steps and search latency.

The second is the accumulation of read-but-unexplored neighbor vectors. When the neighbor exploration is slow, the pre-read neighbors may be accumulated in memory but unexplored. This causes sub-optimal I/O decisions because of missed neighbor information, and thus intrinsic I/O waste of PipeSearch. Compared to best-first search, PipeSearch shows 1.34×/1.43× average I/O per search when $W = 8$.

We raise a question: **Can we reduce the I/O waste in PipeSearch, to achieve low latency and high throughput simultaneously?** Based on the analysis above, we summarize two challenges for this question. *The first* is to dynamically adjust to suitable pipeline widths in a single search. *The second* is to avoid the accumulation of neighbor vectors.

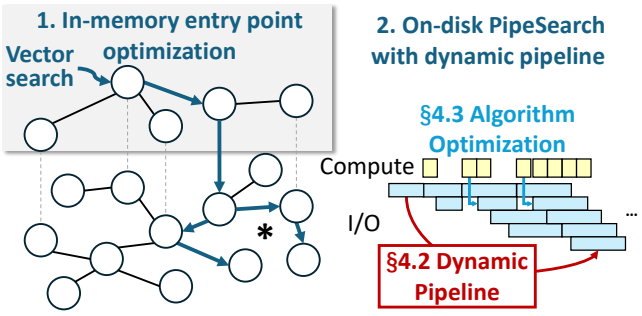


Figure 7: PipeANN overview.

4 PipeANN Design and Implementation

We design PipeANN, a low-latency graph-based ANNS system on disk with high search throughput. PipeANN integrates

Algorithm 2 Optimized PipeSearch in PipeANN

```
1:  G ← graph, q ← query vector, L ← candidate pool length
2:  L_m ← candidate pool length of the in-memory index
3:  procedure PipeSearch(G, q, L)
4:      W ← 4                                   ▷ Starting pipeline width.
5:      // Approach phase: entry point optimization.
6:      candidate pool P ← InMemSearch(q, min(L, L_m))
7:      explored pool E ← ∅
8:      unexplored set U ← ∅, unfinished I/Os Q ← ∅
9:      // Converge phase: dynamic pipeline.
10:     while P ⊊ E do
11:         if Q.size() < W then                ▷ I/O pipeline not full.
12:             V ← top-1 nearest vectors to q in P, not in E
13:             Q.insert(V)                     ▷ send read requests.
14:         end if
15:         v ← nearest vector to q in U        ▷ overlap with I/O.
16:         E.insert(v), U.remove(v)            ▷ explore v.
17:         for nbr in v.neighbors do
18:             dis ← PQ_distance(nbr, q)
19:             P.insert(<nbr, dis>)
20:         end for                             ▷ update candidate pool.
21:         P ← l nearest vectors to q in P     ▷ PQ distance.
22:         F ← finished I/Os in Q              ▷ poll for completion.
23:         W ← AdaptPipelineWidth(P, F)
24:         U.insert(F), Q.remove(F)
25:     end while
26:     return k nearest vectors to q in E      ▷ exact distance.
27: end procedure
```

PipeSearch and improves it using dynamic pipeline width (§4.2) and algorithm optimization (§4.3).

4.1 PipeANN Overview

Figure 7 shows the overview of PipeANN.

Graph layout. On disk, PipeANN stores the graph as adjacent lists (§2.1). In memory, it stores PQ-compressed vectors for neighbor distance comparison and a small graph-based index for entry point optimization.

Vector search. PipeANN uses PipeSearch for vector search but increases its throughput (Algorithm 2). It separates a single vector search into two phases, *approach* phase and *converge* phase, based on the key observation that I/O waste decreases across steps (§4.2).

In the approach phase, the vector search gradually approaches the target vector, when I/O waste is significant and thus PipeSearch is inefficient. Therefore, PipeANN uses an in-memory index (line 6) for entry point optimization, and then starts PipeSearch (lines 7–25) with a small pipeline width.

In the converge phase, the vector search recalls vectors near the target vector, when I/O waste gradually decreases and thus PipeSearch shows efficiency. Thus, PipeANN dynamically increases the pipeline width (line 23) by estimating the number of recalled vectors and the current I/O waste, based on the candidate pool state and the finished I/O.

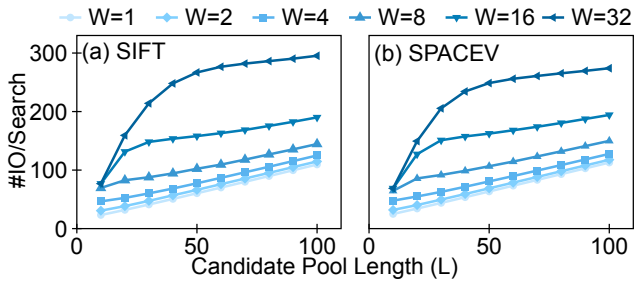


Figure 8: I/O waste of PipeSearch decreases across search steps. In general, it contains a turning point for each W, after which the I/O waste is similar to the ideal case (i.e., $W = 1$).

Also, PipeANN reduces the I/O waste of PipeSearch by not ensuring a perfect pipeline at all times (lines 11–14, §4.3). When multiple I/Os finish simultaneously, PipeANN does not immediately fill up the pipeline. Instead, PipeANN repeatedly issues one I/O and explores one vector in the candidate pool. In this way, it ensures that the nth I/O is decided by the neighbor information of the $(n-W)$th compute, thus increasing the I/O accuracy. The perfect pipeline can be finally ensured, as the I/O completion times tend to differ in this strategy.

4.2 Dynamic Pipeline

PipeSearch fails to ensure low latency and high throughput simultaneously with a fixed pipeline width. Therefore, we propose to adapt the pipeline width dynamically in search. In this section, we first introduce the key observation that motivates us to adapt the pipeline width. Then, we describe the two phases of vector search separately.

4.2.1 I/O Waste Decreases Across Search Steps

In §3.4, we demonstrate the dilemma of PipeSearch in latency and throughput. With higher pipeline width, PipeSearch shows low latency but low throughput, because of high I/O waste. However, we observe that this dilemma is not fundamental: It considers ANNS at a coarse granularity of a whole search, instead of each search step.

To consider ANNS at the granularity of each step, we evaluate the average I/O per search with different Ls, as shown in Figure 8. This evaluation is based on an observation: PipeSearch *with a small L is approximately a subprocess (in other words, an early termination) of one with a large L.* Therefore, the slope in Figure 8 reveals the average I/O for recalling the Lth vector; a smaller slope means less I/O waste.

In Figure 8, we observe that the I/O waste across search steps shows two stages. At the beginning of the search, PipeSearch shows a significant I/O waste, which increases with W. Then, it reaches a turning point, after which the I/O

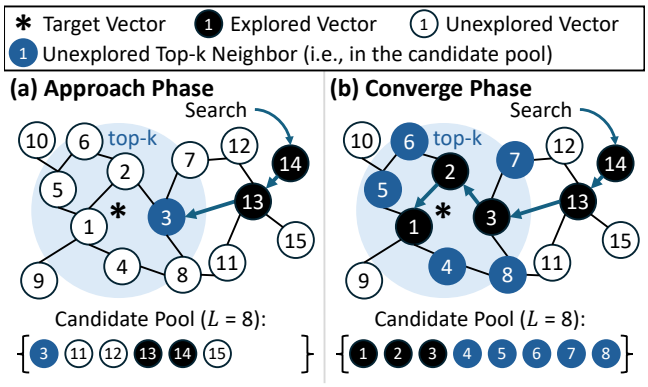


Figure 9: Two-phase search of graph-based ANNS indexes.

waste is similar to the ideal case (i.e., $W = 1$). The turning point arrives later (i.e., a larger L) for a larger W.

We use Figure 9 to further demonstrate this observation. The best-first policy of vector search naturally divides it into two phases, *approach phase* and *converge phase*. In the approach phase, the vectors in the candidate pool quickly approach the target vector. During this phase, PipeSearch with large pipeline widths fails to recall more vectors that are likely in the search's critical path, inducing I/O waste.

In the converge phase, the nearest vector in the candidate pool remains stable, and the top-k nearest vectors are gradually recalled. The more recalled vectors (larger L in Figure 8), the more top-k neighbors they connect. Thus, there are more "unverified" top-k vectors in the candidate pool, which allows using a large pipeline width for fast verification. During this phase, the average I/O for each vector remains stable.

Method: two-phase graph traversal. Based on the observations above, we use different approaches in the two phases, in order to reduce the I/O waste in PipeSearch. We seek to quickly pass the approach phase, where PipeSearch shows inefficiency. Therefore, we optimize the entry point (§4.2.2) and start PipeSearch with a small pipeline width. In the converge phase, we dynamically increase the pipeline width of PipeSearch, based on the candidate pool state (§4.2.3).

Pipeline draining does not last long. This recalls the Observation 2 in §3.1. In the later search steps, most vectors are recalled, so PipeANN fails to ensure a full pipeline. However, as shown in Figure 8, the I/O waste is not significant in the later search steps. We conclude that most remaining vectors are accurate top-k neighbors of the target vector: New vectors are less likely to be inserted into the candidate pool when reading them. Therefore, this process does not last long.

4.2.2 Approach Phase: Entry Point Optimization

PipeANN uses an in-memory graph-based index for entry point optimization, similar to previous works [25]. Specifically, PipeANN samples a portion of entry points in the dataset and builds an in-memory graph-based index using them. The index is built offline and is thus static. For the online vector search, PipeANN first traverses the in-memory index to select entry points and then conducts on-disk PipeSearch.

Parameter selection. In-memory index traversal lies in the critical path of the whole vector search. Therefore, we should carefully select its parameters to ensure its low latency.

Following previous work [25], PipeANN uses a 1% sample rate for the entry points to balance entry point quality and memory usage. We use Vamana (i.e., in-memory DiskANN) as the graph index structure. The index is only used to efficiently find the top-k sampled entry points, so other indexes (e.g., HNSW [16] and NSG [6]) show similar tradeoffs to Vamana. This is also demonstrated in previous work [25].

Here, we discuss the maximum out-degree, which contributes most to the search latency. We build the in-memory index using the Vamana algorithm [23], which is an approximation of a Monotonic Relative Neighborhood Graph (MRNG) [6]. Its search complexity linearly increases with the average out-degree. As this in-memory index is only used for entry point selection, not the overall indexing, we can use a smaller maximum out-degree, tolerating the connectivity loss of some points for lower search latency. By default, we use 32 as the maximum out-degree of in-memory indexes, smaller than in previous works [6, 25].

4.2.3 Converge Phase: Pipeline Width Adjustment

PipeANN dynamically adjusts the pipeline width during the search. To achieve this, two questions should be answered. The first is when to start adjusting the pipeline width, and the second is which pipeline width should be selected.

When to start adjusting. We approximate the number of vectors that are already recalled n_v. When the number reaches a threshold, we consider the search to reach the converge phase and start to increase the pipeline width.

Specifically, after exploring one vector, we iterate over the candidate pool to find the first vector whose read request has not been issued. Its index is used as an approximation of n_v's upper bound. This is not an accurate estimation because there may be unexplored vectors that have been already read. Exploring them may add new nearer neighbors.

We further understand this estimation. In the approach phase, there are usually new nearest neighbors after exploring a vector, so the estimated n_v typically equals 0. In the converge phase, it is hard to find nearer neighbors than the already explored ones by exploring a new vector, so the estimated n_v gradually increases.

After the estimated n_v reaches a threshold (5 in our evaluation), we start adjusting the pipeline width. Before that, the pipeline width is set to a fixed value of 4 to reduce I/O waste.

How to adjust. We propose two approaches for this, static approach and dynamic approach, and we use the dynamic approach by default.

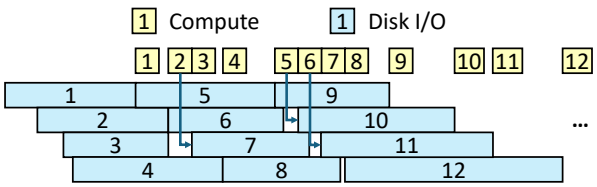


Figure 10: When multiple I/O finish simultaneously, we issue one I/O after exploring one record, instead of issuing multiple I/O to saturate the pipeline. Therefore, every disk I/O misses the neighbor information in no more than W records.

The static approach first profiles the dataset by PipeSearch with different Ws and Ls to get the results like Figure 8. Then, it generates a fixed (#vectors recalled – W) mapping based on the results. In this mapping, each "#vectors recalled" corresponds to the largest W after its turning point. During a vector search, it adjusts W based on the estimated n_v.

The dynamic approach uses another metric: the percentage of I/O that the vector fetched is in the candidate pool. Specifically, after starting pipeline width adjustment, we re-calculate this ratio when there exists finished I/O. When the ratio is greater than a threshold (0.9 in our evaluation), we increase the pipeline width by 1.

4.3 Algorithm Optimization

Compared to beam search, naive PipeSearch shows a throughput drop, especially with large pipeline widths, due to I/O waste caused by accumulated neighbor vectors. Here, we describe this in detail.

The root cause of I/O waste is missing neighbor information in the fetched vectors. Less missed neighbor information induces more accurate I/O decisions, thus less I/O waste. For example, greedy search issues I/O after exploring all the vectors, so it misses no neighbor information and shows less I/O waste compared to beam search and PipeSearch.

Ideally, if compute and I/O are perfectly overlapped, each I/O in PipeSearch misses the neighbor information in no more than W records (in W on-flight I/Os). However, this is based on the assumption that I/Os are finished uniformly in the timeline, which is not true in SSDs.

When multiple I/Os finish simultaneously, filling up the pipeline may make I/O miss the neighbor information in more records (W on-flight I/Os and multiple read-but-unexplored vectors). To avoid this, we limit the I/O rate by handling multiple I/Os one by one, as shown in Figure 10. Specifically, after multiple I/O finishes, we repeatedly send one I/O, explore one nearest vector, and update the neighbor set. This way can reduce I/O waste and gradually scatter the I/Os uniformly in the timeline, thus not incurring much overhead.

4.4 Implementation and Other Optimizations

Overlapping initialization. PipeANN needs to wait for the first disk I/O to finish, to get neighbor information for filling up the pipeline. This process has a latency of $\sim 50\mu$s in our NVMe SSD. We overlap it with local PQ table initialization, which is performed for each query to enable fast PQ distance lookup. However, local PQ table initialization needs to read the global PQ table, which is not used in search and thus causes cache pollution. We use non-temporal load in AVX512 instruction to avoid this issue.

Asynchronous I/O. PipeANN uses `io_uring` [11] to issue I/O requests, due to its performance and compatibility. Specifically, each thread uses its private `io_uring` to send I/O requests asynchronously, with the `prep_read` command (line 13 in Algorithm 2). It polls for I/O completion (line 22) using the non-blocking `peek_batch_cqe` command.

Polling-based I/O. Existing systems adopt interrupt-based I/O for best-first search. They do not use polling-based I/O as its latency advantage does not boost the performance. They synchronously wait for all the I/Os in a batch to finish before compute, thus interrupt overhead is minor compared to I/O latency. However, PipeANN needs to issue and poll I/O with low latency, as it could use the saved I/O time for computation. Hence, we enable SQ polling of the `io_uring` engine.

4.5 Discussion

Beyond SSD. PipeSearch is designed for on-disk graph-based ANNS but is not restricted to it. The mechanisms can be used for other hardware with μs-scale access latency, such as remote memory using remote direct memory access (RDMA) or compute express link (CXL). In this case, the whole graph-based index is stored in remote memory. PQ-compressed vectors and the small index for entry point optimization are stored in local memory. Asynchronous I/O could be implemented by replacing `io_uring` commands with corresponding remote memory commands (e.g., using RDMA read or CXL prefetch instead of `prep_read`). As the remote memory latency (e.g., 2μs for RDMA) is still the same order of magnitude with compute (μs-scale for each record), PipeSearch is expected to boost ANNS performance on remote memory.

Two-phase search in other works. VBASE [31] also observes a similar phenomenon of two-phase vector indexing, called relaxed monotonicity. It leverages this for efficient similarity search with tags, by designing an interface that returns the next vector in replace of the interface that returns top-k vectors. We exploit this phenomenon to reduce I/O waste in PipeSearch, aiming to ensure both low latency and high throughput. Specifically, we observe that I/O waste decreases during the search process, given a fixed pipeline width.

Memory usage. PipeANN requires <40GB of memory for billion-scale datasets, including:

- 32GB for PQ-compressed vectors, 32 bytes per vector.

- <4GB for the in-memory graph index. In our evaluation, it uses 2.4GB for the SIFT1B [10] dataset and 3.1GB for the SPACEV1B dataset [4].
- Minor overheads for other in-memory data structures.

In comparison, DiskANN [23] requires ~32GB of memory, mainly for the PQ-compressed vectors.

PipeANN's memory-to-disk size ratio is ~1:15, considering that billion-scale graph indexes on disk typically require over 600GB of disk space (e.g., 636GB for SIFT1B and 892GB for SPACEV1B). PipeANN's primary memory constraint lies in the PQ-compressed vectors, similar to DiskANN. Therefore, memory-efficient quantization (e.g., RaBitQ [7]) methods could reduce their memory usage.

5 Evaluation

In the evaluation, we seek to answer the following questions:

- How does PipeANN perform in latency and throughput compared to other on-disk ANNS indexes? (§5.2)
- Can PipeANN scale to billion-scale datasets? (§5.3)
- Is PipeANN comparable with the in-memory graph-based ANNS index in terms of search latency? (§5.4)
- How do the techniques in PipeANN contribute to its performance? (§5.5)
- How do the approaches for pipeline width adjustment impact search performance? (§5.6)
- How much does PipeANN trades throughput and accuracy with the same search parameters for latency? (§5.7)

5.1 Experimental Setup

Basic configuration. We use one server for evaluation, which has the following configuration:

- **CPU**: 2× 28-core Intel Xeon Gold 6330 @ 2.00GHz;
- **RAM**: 512GB (16× 32GB DDR4 2933MT/s);
- **SSD**: 1× Samsung PM9A3 3.84TB;
- **OS**: Ubuntu 22.04 LTS with Linux kernel 5.15.0.

Compared systems. We compare PipeANN with on-disk ANNS indexes, including graph-based DiskANN [23] and Starling [25], as well as cluster-based SPANN [4]. DiskANN is a graph-based ANNS index using best-first search. Starling optimizes the I/O of DiskANN by reordering the on-disk records to improve search locality and using an in-memory index to optimize the entry point.

SPANN is an on-disk cluster-based index. It separates the vectors into clusters and maintains an in-memory navigation graph to index the cluster centroids. To conduct ANNS, it first searches the nearest clusters in the in-memory graph and then re-ranks all the vectors in the clusters to get the results.

Parameters. We use the same in-memory and on-disk graph indexes for PipeANN, DiskANN, and Starling, except that Starling reorders the on-disk records. The graph indexes are built using the parameters in Table 1. For search parameters, the candidate pool size of in-memory index traversal L_{mem} is fixed to 10 for PipeANN and Starling. We use `io_uring` [11] as the I/O engine of PipeANN and enable SQ polling. For fairness, we also replace the original I/O engine (`libaio`) with `io_uring` for DiskANN and Starling. We disable SQ polling for them. We use $W = 8$ for DiskANN and $W = 4$ for Starling, which shows the lowest latency separately. We limit the maximum W of PipeANN to 32.

Graph Type	R	L	B
On-Disk (100M)	96	128	32
On-Disk (1B)	128	200	32
In-Memory	32	64	/

Table 1: Parameters used for Vamana (i.e., DiskANN) graph building [23]. R: maximum out-degree. L: candidate pool size for finding neighbors. B: PQ-compressed vector size (bits/vector). The in-memory index is used by PipeANN and Starling for entry point optimization.

Name	#Vectors	Type	Dim	#Queries
SIFT1B [10]	1B	`uint8`	128	10,000
SPACEV1B [4]	1.4B	`int8`	100	29,316
SIFT100M [10]	100M	`uint8`	128	10,000
DEEP100M [2]	100M	`float`	96	10,000
SPACEV100M [4]	100M	`int8`	100	29,316

Table 2: Datasets used in the evaluation. SIFT100M, DEEP100M, and SPACEV100M are subsets of SIFT1B, DEEP1B, and SPACEV1B.

For SPANN, we build cluster-based indexes using the same parameter in the SPFresh [29] repository. Specifically, for SIFT and SPACEV, we set their maximum cluster size to 16KB. For DEEP, we set it to 48KB. Each vector is replicated to its nearest 8 clusters.

Datasets. We use five public datasets in the evaluation, including 100M-scale and billion-scale datasets. Detailed configurations of the datasets are shown in Table 2.

Metrics. We mainly compare the latency and throughput for 0.9 recall, which is recommended by the BigANN benchmark [20]. Low recall (e.g., 0.8) and high recall (e.g., 0.99) are also used in some experiments for more thorough comparison. We evaluate the recall by searching the top 10 nearest neighbors (i.e., recall10@10).

5.2 Overall Performance

In this section, we evaluate the latency and throughput of PipeANN using datasets with 100M vectors. We do not use billion-scale datasets, where the graph reordering algorithm in Starling has huge time and memory overhead. We will

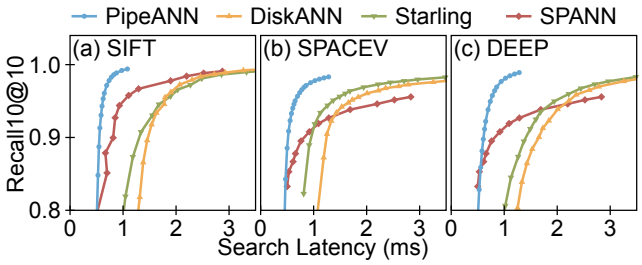


Figure 11: Search latency on datasets with 100M vectors.

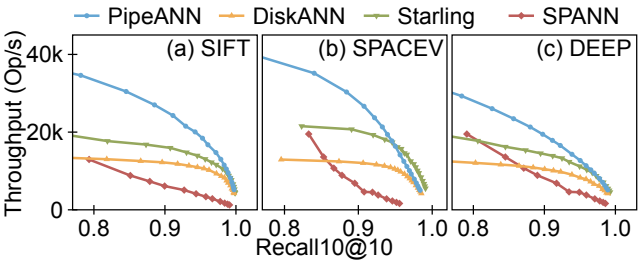


Figure 12: Search throughput on datasets with 100M vectors.

show the results in billion-scale datasets in §5.3.

5.2.1 Latency

We use 1 thread to conduct ANNS in the compared systems. Figure 11 shows the results. From the figure, we make the following observations:

(1) Compared with graph-based indexes, PipeANN shows lower latency. To achieve 0.9 recall10@10, PipeANN has 39.1%/48.5% latency on average, compared to DiskANN/Starling. This is because PipeANN efficiently uses dynamic PipeSearch to overlap compute and I/O and increase the average I/O depth, thus reducing latency. Starling has lower latency compared to DiskANN. This is because Starling uses the in-memory index for entry point optimization and reorders the on-disk graph to reduce the average I/O per search.

(2) Compared with cluster-based indexes, PipeANN shows lower latency when recall $\geq$ 0.9. To achieve 0.9 recall10@10, PipeANN has 70.6% lower latency compared to SPANN. SPANN shows lower latency than other graph-based indexes, because of its I/O friendliness. SPANN first traverses an in-memory graph-based index for the nearest clusters of the target vector. Then, it sends I/O to fetch the clusters in parallel. In contrast, existing graph-based indexes traverse the graph by issuing I/O batches in sequential, where I/O latency significantly harms the overall search latency. PipeANN greatly eliminates this issue by pipelining, which makes it faster than SPANN for recall $\geq$ 0.9. However, when the recall is smaller (e.g., 0.8), PipeANN suffers from the overhead of approaching the target vector, making it slower than SPANN.

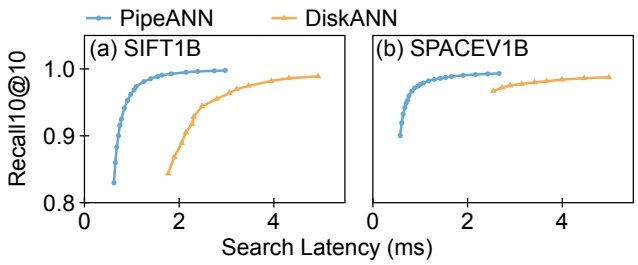


Figure 13: Search latency in billion-scale datasets.

5.2.2 Throughput

We use 56 threads (all the cores of our CPU) to conduct ANNS in the compared systems. Figure 12 shows the results. From the figure, we make the following observations:

(1) When recall = 0.9, PipeANN consistently shows the highest throughput. To achieve 0.9 recall10@10, PipeANN outperforms other systems by 1.35× on average. In this case, the disk bandwidth is not saturated, because other tasks (e.g., PQ table initialization) account for a high time percentage. The extra I/O of PipeANN does not harm the overall throughput much. PipeANN's high throughput owes to its shorter critical path, because of pipelining.

(2) For higher recall, PipeANN has a lower throughput than Starling. To achieve 0.99 recall10@10, PipeANN has 0.80× lower throughput than Starling on average, because of the wasted disk I/O. In this case, the search procedure takes up most of the time, which saturates the disk bandwidth. For each search, PipeANN requires 1.94× average disk I/O on average compared to Starling, which results in lower throughput. This is because of the reordering technique of Starling. Starling reorders the on-disk index for more neighboring records on the same page, thus reducing disk I/O.

Note that the design of PipeANN is orthogonal with Starling. In PipeANN, we can directly adopt the same reordering technique to reduce I/O. However, considering the huge time and memory overhead for reordering in billion-scale datasets, we do not adopt this technique in PipeANN.

Compared with DiskANN, PipeANN shows higher throughput. PipeANN has a similar (0.98× for 0.99 recall) average I/O per search compared to DiskANN, but it can better utilize the disk bandwidth because of pipelining. This results in PipeANN's higher throughput.

5.3 Overall Performance: Billion-Scale

In this section, we show the performance of PipeANN in billion-scale datasets. We compare PipeANN with DiskANN. We do not compare PipeANN to other baselines because building or searching them exceeds the memory capacity in our setup. Figures 13 and 14 show the results. In SPACEV1B, we do not show DiskANN with accuracies less than 96%,

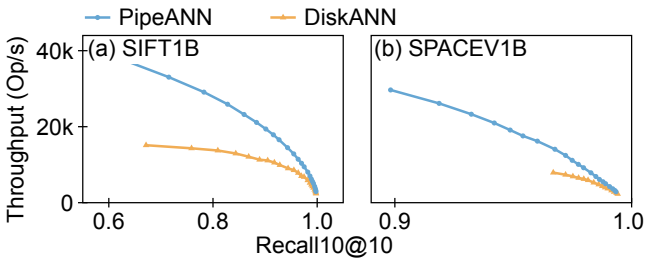


Figure 14: Search throughput in billion-scale datasets.

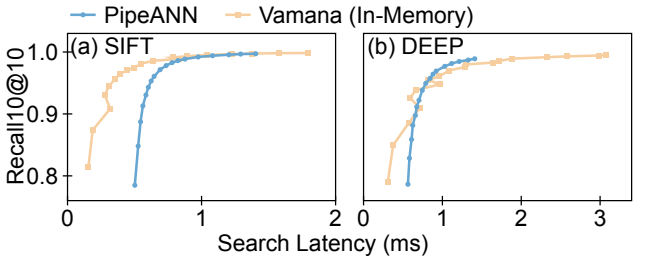


Figure 15: Search latency of PipeANN compared with Vamana (in-memory DiskANN).

because its accuracy only achieves 60% using a smaller L.

Latency. As shown in Figure 13, to achieve 0.9 recall10@10, PipeANN has latencies of 0.719 ms and 0.578 ms in SIFT1B and SPACEV1B, which are 1.28×/1.09× compared to SIFT100M and SPACEV100M. Compared to DiskANN, PipeANN achieves 35.0% latency in SIFT.

Throughput. As shown in Figure 14, to achieve 0.9 recall10@10, PipeANN has 19.4K and 26.1K QPS in SIFT1B and SPACEV1B, which are 79.9% and 98.0% compared to SIFT100M and SPACEV100M. Compared to DiskANN, PipeANN achieves 1.71× higher throughput.

Analysis. The search path is longer for billion-scale datasets than 100M-scale datasets, which incurs a higher search latency to achieve the same recall. However, it also brings more opportunities for pipelining, which contributes to a more significant performance boost compared to DiskANN.

5.4 Compare with In-Memory Index

In this section, we show the performance gap of PipeANN with an in-memory graph-based index. We directly store the on-disk index of PipeANN in memory and use it for searching. We call this baseline Vamana. We compare PipeANN with Vamana using SIFT100M and DEEP100M datasets. Figure 15 shows the results, and we make the following observations:

(1) For low recall, PipeANN shows higher latency than the in-memory index. To achieve 0.8 recall10@10 in SIFT100M, PipeANN has 3.38× higher latency than Vamana. Because of a small L (e.g., 10), the search is mainly in the approach phase, where PipeSearch cannot maintain a large pipeline width. Therefore, PipeANN fails to hide all the I/O latency, which makes the performance of PipeANN lower than the in-memory index.

(2) For recall ≥ 0.9, the performance of PipeANN is close to the in-memory index. To achieve 0.9 recall10@10, PipeANN has 2.02×/1.14× latency compared to Vamana. In this case, both PipeANN and Vamana use a large L (e.g., 30). The converge phase accounts for a higher ratio, where PipeSearch maintains a large pipeline width for better overlapping of compute and I/O. Therefore, the performance gap between PipeANN and Vamana reduces.

PipeANN's slowdown is less significant in DEEP100M. This is because, in DEEP, distance comparison for Vamana is more costly than in SIFT. DEEP calculates 96 `floats` per vector, which has a higher overhead than the 128 `uint8`s per vector in SIFT. In contrast, PipeANN uses PQ distance for neighbors, which has similar overheads for distance comparison in the two datasets. We conclude that PipeANN is expected to have more comparable performance with in-memory ANNS with higher vector dimensions.

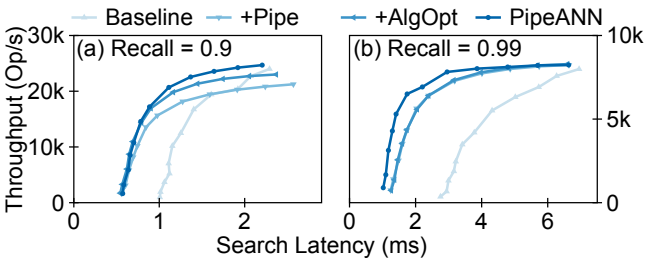


Figure 16: Breakdown analysis of PipeANN.

5.5 Breakdown Analysis

In this section, we break down the performance gap of Baseline and PipeANN. We accumulate key techniques into the Baseline and evaluate the latency-throughput graph with 0.9 and 0.99 recall10@10. We use the index built on the SIFT100M dataset, and the same configurations as in §5.2.

Baseline. We implement the Baseline based on the framework of PipeANN. It uses best-first search like DiskANN with $W = 8$ and adopts the same in-memory index as PipeANN for entry point optimization. As shown in Figure 16(a), the Baseline shows ms-scale latency to achieve 0.9 recall. Although adopted entry point optimization, its best-first search algorithm still incurs high search latency on disk.

+Pipe. We use PipeSearch (§3) with $W = 8$ in replace of best-first search in Baseline. It reduces latency to 55.1% for 0.9 recall. However, the throughput is also reduced to 88.5%. This is because PipeSearch has a 1.11× average I/O per search compared to best-first search. The I/O waste and throughput

#Vectors Recalled	0	10	20	30	40
Pipeline Width	4	8	16	24	32

Table 3: Pipeline width used by static approach.

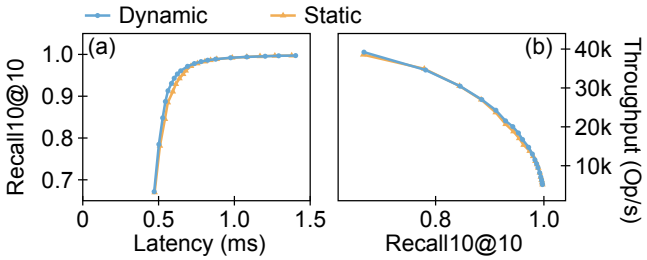


Figure 17: Performance of dynamic approach and static approach for pipeline width adjustment. Dataset: SIFT100M.

drop are less significant than in §3.4, as entry point optimization reduces the I/O waste during the approach phase.

+AlgOpt. We adopt the algorithm optimization for multiple finished I/Os (§4.3). It increases the throughput to 1.08× because it reduces the average I/O per search to 91.8%. The performance boost is more significant in high-throughput cases, which concludes that multiple I/Os are more frequent to finish simultaneously in such cases. The latency is slightly reduced to 97.5%, as the performance degradation of the unsaturated I/O pipeline is less significant than the performance boost of reduced I/O.

PipeANN. We adopt the dynamic pipeline (§4.2) in replace of the static one with $W = 8$. It reduces the latency to 81.1% for 0.99 recall and increases the throughput to 1.07×. This is because, in the converge phase, a large W allows more compute and I/O overlapping, while not inducing much I/O waste. It does not show much performance boost for 0.9 recall, as the search terminates early in the converge phase when W is not significantly larger than 8.

5.6 Approaches for Pipeline Adjustment

In this section, we compare the two approaches for pipeline width adjustment (§4.2.3), namely the static approach and the dynamic approach. For the static approach, we set the pipeline width according to the profiling results similar to Figure 8(a). The parameters used are shown in Table 3.

Figure 17 shows the results. We find that the dynamic approach slightly outperforms the static approach by up to 6.1%/9.1% for latency and throughput. This demonstrates that PipeANN is not sensitive to the approach for pipeline width adjustment. The dynamic approach successfully follows the decreasing trend of I/O waste to increase the pipeline width across the search steps.

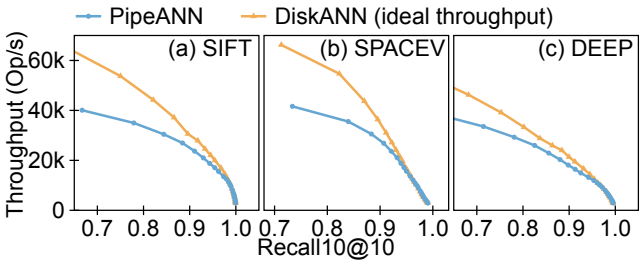


Figure 18: Throughput of PipeANN and DiskANN with ideal throughput (best-first search with $W = 1$).

5.7 Tradeoffs in PipeANN

PipeANN tweaks the best-first search algorithm for lower latency, but with some tradeoffs. In this section, we demonstrate them using experiments.

Throughput. Although PipeANN introduces techniques to reduce I/O waste, its $W > 1$ could still waste more I/O compared to the ideal best-first search with $W = 1$ (note that we use $W = 8$ for lower latency in §5.2), and thus lead to lower throughput. We implement this ideal baseline using DiskANN, which we call DiskANN (ideal throughput). To saturate SSD, this baseline uses asynchronous I/O; one CPU core executes multiple search requests simultaneously and switches across them to avoid I/O waiting. We compare PipeANN with this baseline using the same datasets as in §5.2.

Figure 18 shows the results. From the figure, we observe that: First, PipeANN shows lower throughput than the ideal DiskANN at low accuracy. When recall = 0.8, the throughput drop of PipeANN is 31.6%/34.1%/17.5% separately in the three datasets. This is because the approach phase accounts for a large percentage, when PipeANN's $W > 1$ leads to I/O waste. Second, the throughput drop becomes less significant at higher accuracy. When the recall reaches 0.95, the throughput drop becomes 14.7%/6.15%/4.90%. In such cases, the converge phase lasts longer, and some wasted I/O in the approach phase are also explored in best-first search. These two reasons contribute to less I/O waste of PipeANN. Third, at the same accuracy, PipeANN shows less I/O waste in DEEP than the other two datasets. This is because DEEP requires a larger candidate pool length L, and thus a longer converge phase. This leads to less I/O waste.

Search accuracy with the same parameters. PipeANN tweaks the best-first search algorithm, which may lead to an accuracy drop under the same search parameters. Here, we evaluate this issue in terms of quantity. We compare PipeANN with the same L as DiskANN, which uses best-first search with $W = 8$. Figure 19 shows the results. From the figure, we make the following observations:

(1) PipeANN shows little accuracy drop. PipeANN has at least 95.9% recall compared to DiskANN. When recall ≥ 0.9, this value is further increased to 98.8%. The accuracy drop

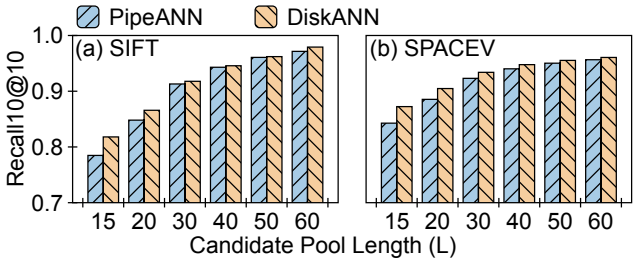


Figure 19: Accuracy of PipeANN and DiskANN with the same L. Note that the Y-axis does not start at zero.

is because PipeANN approximates the best-first search algorithm. However, the modification is slight enough to ensure similar behavior for search convergence.

(2) The accuracy drop is less significant for larger L. We understand this by regarding PipeANN as a "best-first search" with a candidate pool length $L-W$, noticing that each I/O in PipeANN misses at most W vectors. The search accuracy increases slower for larger L, so the accuracy gap of best-first searches with $L-W$ and L becomes less significant.

PipeANN achieves the same accuracy with DiskANN using a larger L, but benefits from lower latency.

6 Related Work

PipeANN targets on-disk graph-based ANNS with low latency. In this section, we introduce two types of related work, namely graph search optimization and on-disk ANNS.

Graph search optimization. Traditional graph-based indexes do ANNS in memory. They typically use greedy search to traverse from the starting vector to the target vector [6, 16]. Compared to distance comparison, memory access for vectors has orders of magnitude lower latency, so the greedy-based approach is favored for reduced vectors accessed per search. In such a scenario, distance comparison is the major bottleneck of graph-based ANNS, so some works focus on reducing the overhead of distance comparison for high performance [3, 30].

On disk, the best-first algorithm incurs high search latency when exploring one vector at a time (i.e., $W = 1$). Therefore, DiskANN [23] proposes to use beam search, where it reads the best k vectors in parallel for each search step. Beam search is also used by iQAN [18] to do intra-query parallelism among multiple cores, where each core explores one vector. Compared to greedy search, beam search reduces the latency but still follows a best-first algorithm, which induces compute-I/O order and limits the performance.

Some other works accelerate graph search by reducing the length of the search path, using entry point optimization and early termination. LSH-APG [32] and Starling [25] optimize the fixed entry point using an LSH table or a sub-graph. Proxima [28] and learned adaptive early termination [13] determine the search termination condition based on the search state and an ML model, separately. The design of PipeANN is orthogonal to these works. These works can be directly adopted by PipeANN for acceleration.

In contrast, PipeANN reduces the search latency by aligning the best-first search algorithm with SSD characteristics.

On-disk ANNS. To support larger datasets, recent works store ANNS indexes on disk. They can be mainly divided into two types, graph-based indexes and cluster-based indexes.

Graph-based indexes are favored for their search accuracy and I/O efficiency, which are suitable for high-throughput ANNS. They typically store compressed vectors in memory for navigation, and the full index on disk [8, 23, 25]. However, due to the best-first search algorithm, the critical path in search includes tens of sequential I/O. Therefore, graph-based indexes suffer from high search latency on disk, which motivates the designs of PipeANN. Also, updating the graph-based indexes induces high overheads [22, 29].

Cluster-based indexes [4, 24] have a simpler structure than graph-based indexes. They divide the vectors into multiple clusters, each of which contains tens of vectors. The clusters are stored on disk, and indexed by an in-memory graph. Vector search first finds the nearest clusters in memory and then reads them on disk in parallel. Therefore, only one parallel disk I/O lies in the critical path, which contributes to its low search latency. Also, updating the cluster-based indexes incurs a lower cost than graph-based ones [29]. However, cluster-based indexes are more coarse-grained than graph-based ones, which induces lower search throughput.

7 Conclusion

We propose PipeSearch, a low-latency algorithm for on-disk graph-based ANNS. PipeSearch accelerates on-disk ANNS by aligning the best-first search algorithm with SSD features. It benefits from compute-I/O overlapping and a better-utilized I/O pipeline compared to best-first search. We optimize PipeSearch and implement PipeANN, a low-latency ANNS system with high search throughput. Experiments show that PipeANN significantly bridges the latency gap between in-memory ANNS and on-disk ANNS. This work demonstrates that aligning the algorithm with the hardware characteristics brings performance benefits.

Acknowledgments

We sincerely thank our shepherd, Patrick P. C. Lee, and the anonymous reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (Grant No. 2024YFE0203300), the National Natural Science Foundation of China (Grant No. 62332011), and Beijing Natural Science Foundation (Grant No. L242016).

References

[1] Ilias Azizi, Karima Echihabi, and Themis Palpanas. Graph-Based Vector Search: An Experimental Evaluation of the State-of-the-Art. In *Proceedings of the ACM on Management of Data*, SIGMOD '25, Berlin, Germany, 2025. Association for Computing Machinery.

[2] Artem Babenko and Victor S. Lempitsky. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, pages 2055–2063, Las Vegas, NV, USA, 2016. IEEE Computer Society.

[3] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. FINGER: Fast Inference for Graph-based Approximate Nearest Neighbor Search. In *Proceedings of the ACM Web Conference 2023*, WWW '23, pages 3225–3235, Austin, TX, USA, 2023. Association for Computing Machinery.

[4] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: highly-efficient billion-scale approximate nearest neighbor search. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc.

[5] Rongxin Chen, Yifan Peng, Xingda Wei, Hongrui Xie, Rong Chen, Sijie Shen, and Haibo Chen. Characterizing the Dilemma of Performance and Index Size in Billion-Scale Vector Search and Breaking It with Second-Tier Memory. *CoRR*, abs/2405.03267, 2024.

[6] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *Proceedings of the VLDB Endowment*, VLDB '19, pages 461–474, Los Angeles, CA, USA, 2019. VLDB Endowment.

[7] Jianyang Gao and Cheng Long. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. In *Proceedings of the ACM on Management of Data*, SIGMOD '24, Santiago, Chile, 2024. Association for Computing Machinery.

[8] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*, WWW '23, pages 3406–3416, Austin, TX, USA, 2023. Association for Computing Machinery.

[9] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, Dallas, TX, USA, 1998. Association for Computing Machinery.

[10] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: rerank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '11, pages 861–864, Prague, Czech Republic, 2011. IEEE Computer Society.

[11] Kanchan Joshi, Anuj Gupta, Javier Gonzalez, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux. In *22nd USENIX Conference on File and Storage Technologies*, FAST '24, pages 107–121, Santa Clara, CA, 2024. USENIX Association.

[12] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[13] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2539–2554, Portland, OR, USA, 2020. Association for Computing Machinery.

[14] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 9–16, Rennes, France, 2018. Association for Computing Machinery.

[15] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based Product Retrieval in Taobao Search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pages 3181–3189, Virtual Event, 2021. Association for Computing Machinery.

[16] Yu A. Malkov and D. A. Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions*

on Pattern Analysis and Machine Intelligence (TPAMI), 42(4):824–836, 2020.

[17] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, Workshop Track Proceedings*, ICLR '13, Scottsdale, Arizona, USA, 2013.

[18] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. iQAN: Fast and Accurate Vector Search with Efficient Intra-Query Parallelism on Multi-Core Architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, pages 313–328, Montreal, QC, Canada, 2023. Association for Computing Machinery.

[19] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning*, ICML '21, pages 8748–8763, Virtual Event, 2021. PMLR.

[20] Harsha Simhadri. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc.

[21] Harsha Simhadri. Research talk: Approximate nearest neighbor search systems at scale. https://www.youtube.com/watch?v=BnYNdSIKibQ&list=PLD7HFcN7LXReJTWFKYqwMcCc1nZKIXBo9&index=9, 2022.

[22] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *CoRR*, abs/2105.09613, 2021.

[23] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, NIPS '19, Red Hook, NY, USA, 2019. Curran Associates Inc.

[24] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs. In *2024 USENIX Annual Technical Conference*, USENIX ATC '24, pages 1135–1150, Santa Clara, CA, 2024. USENIX Association.

[25] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. In *Proceedings of the ACM on Management of Data*, SIGMOD '24, Santiago, Chile, 2024. Association for Computing Machinery.

[26] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. In *Proceedings of the VLDB Endowment*, VLDB '21, pages 1964–1978, Copenhagen, Denmark, 2021. VLDB Endowment.

[27] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. In *Proceedings of the VLDB Endowment*, VLDB '20, pages 3152–3165, Tokyo, Japan, 2020. VLDB Endowment.

[28] Weihong Xu, Junwei Chen, Po-Kai Hsu, Jaeyoung Kang, Minxuan Zhou, Sumukh Pinge, Shimeng Yu, and Tajana Rosing. Proxima: Near-storage Acceleration for Graph-based Approximate Nearest Neighbor Search in 3D NAND. *CoRR*, abs/2312.04257, 2023.

[29] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 545–561, Koblenz, Germany, 2023. Association for Computing Machinery.

[30] Mingyu Yang, Jiabao Jin, Xiangyu Wang, Zhitao Shen, Wei Jia, Wentao Li, and Wei Wang. Effective and General Distance Computation for Approximate Nearest Neighbor Search. *CoRR*, abs/2404.16322, 2024.

[31] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 377–395, Boston, MA, USA, 2023. USENIX Association.

[32] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. In *Proceedings of the VLDB Endowment*, VLDB '23, pages 1979–1991, Vancouver, Canada, 2023. VLDB Endowment.

[33] Xiaoyao Zhong, Haotian Li, Jiabao Jin, Mingyu Yang, Deming Chu, Xiangyu Wang, Zhitao Shen, Wei Jia, George Gu, Yi Xie, Xuemin Lin, Heng Tao Shen, Jingkuan Song, and Peng Cheng. VSAG: An Optimized Search Framework for Graph-based Approximate Nearest Neighbor Search. *CoRR*, abs/2503.17911, 2025.

Skybridge: Bounded Staleness for Distributed Caches

Robert Lyerly
Meta Platforms Inc.

Scott Pruett*

Kevin Doherty
Meta Platforms Inc.

Greg Rogers
Meta Platforms Inc.

Nathan Bronson*
OpenAI

John Hugg
Meta Platforms Inc.

Abstract

Meta Platforms Inc. is a social media company whose products require high availability and low latency. Meta's services run in multiple geographic locations around the world and use asynchronous replication to keep the numerous cached copies of the datastore in sync. This setup reduces consistency in order to meet availability and latency requirements. Eventual consistency due to asynchronous replication causes issues for Meta's services, ranging from minor annoyances to product-breaking bugs. Therefore, we ask: *can we put meaningful bounds on how long it takes writes to be visible while maintaining the scalability afforded by eventual consistency?*

In this work we present Skybridge, an out-of-band replication stream for providing bounded staleness for distributed caches. Skybridge takes advantage of the fact that Meta's systems already have a reliable delivery stream and instead focuses on real-time delivery of updates. Skybridge is complementary to the main replication pipeline and avoids correlated failures while being lightweight. We show that Skybridge helps provide 2-second bounded staleness for 99.99998% of writes, while the main replication pipeline only achieves this 99.993% of the time. Skybridge is able to achieve this while only being 0.54% the size of cache deployments.

1 Introduction

Meta Platforms, Inc. is a social media company whose products are used by people around the world. Like other web-scale serving stacks, Meta's datastore hosts millions of database shards and handles billions of queries per second at sub-second latency [11, 15, 19, 45, 63]. Because of these stringent requirements, Meta operates instances of the serving stack in multiple geographic regions worldwide. Each region stores numerous copies of the data in the form of database (DB) replicas [19, 41, 44], in-memory caches [11, 22, 52, 63] and indexing systems [9, 57]. This architecture provides high scalability and close proximity to users for low latency.

While this architecture is highly scalable, it means there are dozens of copies of millions of shards around the world that are kept up to date. Meta uses asynchronous replication [15] to keep copies synchronized in order to meet our strict latency and high scalability requirements. Asynchronous replication provides *eventual consistency* [6] for clients, meaning all updates will eventually be reflected in all replicas but without bounds on how long it may take. Eventual consistency is problematic when there is non-trivial replication lag that delays when updates are visible to clients. Due to the scale of Meta's serving stack (both in traffic and geographic size), it is almost impossible to eliminate all sources of lag. An overloaded publisher, a hotly-written shard or a congested network hop can all cause replication lag for multiple shards and many downstream replicas.

Eventual consistency is a recurring pain point in many previous works [2, 39, 57]. For Meta, eventual consistency causes bad user experiences in products. Visibility controls are a prime example – for instance, *Alice* may add *Bob* to a closed group. However if the group's membership data is read from a lagging replica, *Bob* will not be able to participate in the group until the replica catches up. This violates *Alice* and *Bob*'s user expectations and causes frustration with our products. Eventual consistency can also have more serious side-effects. For example, Meta's serving stack allow triggering asynchronous actions outside of web request handling to avoid blocking when returning a response. Asynchronous actions may read from different replicas or even execute in different regions. In one instance a content enforcement system using asynchronous actions experienced an outage due to sustained replication lag. The action could not see data written during the web request, leading to a surge of retries that clogged up execution queues and caused system downtime.

Meta's product developers are aware there is replication lag in the system, but without stronger guarantees they build workarounds to deal with distributed systems issues that leak through datastore abstractions. Developers often ask "what is the maximum cross-region replication lag?" Infrastructure teams struggle to provide useful answers as lag can vary from

*Work done while at Meta Platforms Inc.

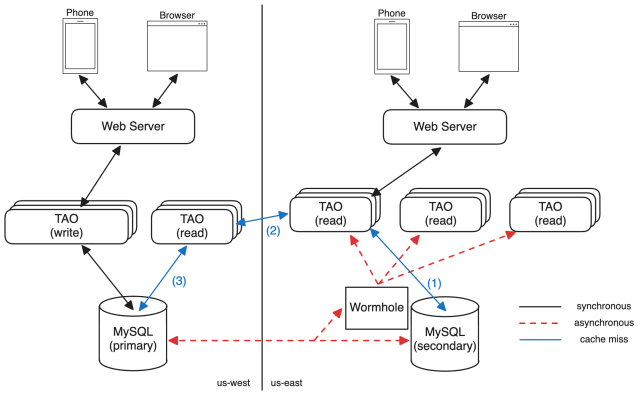


Figure 1: System architecture for Meta's serving stack.

milliseconds to hours (or longer). Developers cope with lag via patterns like retry loops, racy state machines and buggy distributed locks. For example, one developer added a retry loop that would poll the datastore for up to 40 minutes waiting for data. Better guarantees on when writes become visible obviate a whole class of developer issues. Thus, we ask *how can we put reliable and tight bounds on how long it takes for a write to be visible while maintaining the serving stack's high scalability and low latency?*

In this work we present Skybridge, an out-of-band replication system for providing 2-second bounds on eventual consistency. Meta's existing replication system, which provides strong durability and ordering guarantees, already replicates updates within 2 seconds in most cases. Skybridge instead targets the long-tail of lagging shards, where the strong guarantees of traditional replication systems cause replication lag. Our core insight is that Skybridge can implement replication with weaker guarantees while still enabling the system to holistically enforce staleness bounds. The weaker semantic, named *replication with gap detection*, allows out-of-order replication and known data loss. These properties allow Skybridge to side-step common sources of replication lag and focus on real-time replication, complementing the existing replication stream. While Skybridge builds upon existing distributed systems mechanisms, it combines them to provide a new replication guarantee and occupies a novel point in the replication system design space.

Skybridge is designed to be used by in-memory distributed caches, including TAO [11], the cache fronting Meta's serving stack. When the replication stream is lagging, TAO consults Skybridge to determine which individual cache items are stale and need refilling from the source of truth. Skybridge's protections are enabled by default for all TAO requests, transparently boosting consistency with minimal overhead.

In this work we make the following contributions:

- Explain why bounded staleness is important for TAO's users and why we did not target other forms of consistency,
- Define a novel replication guarantee called replication with gap detection that focuses on detecting cache staleness,
- Describe Skybridge's design and implementation, a system that provides fine-grained and efficient bounded staleness for distributed in-memory caches,
- Evaluate Skybridge's ability to provide reliable bounded staleness for TAO users, including demonstrating that TAO's 2-second consistency increases from 99.993% to 99.99998% using Skybridge. Skybridge helps TAO prove data is fresh without having to refill from the primary source for 99.9996% of requests while only requiring 0.54% of TAO's server footprint.

2 Background

2.1 Meta's system architecture

Meta's serving stack is similar to other web serving stacks like PNUTs [15] or Espresso [51]. The serving stack prioritizes high availability and low latency, meaning Meta operates many deployments around the world.

Storage. Meta stores its user data in MySQL [43]. The database is partitioned into millions of shards and replicated across regions. Meta uses primary-secondary replication [44] to keep regions up to date. For each shard, one region hosts the *primary* replica that accepts both read and write requests. Other regions host *secondary* replicas, which only accept read requests. Writes to the primary are *asynchronously* replicated to secondaries, i.e., MySQL acknowledges completed writes to the client before the change is guaranteed to have been replicated to all secondaries.

Caching. Meta's workload is skewed towards reads. This workload is amenable to caching, which allows scaling throughput with low latency. Meta operates an in-memory caching service named TAO [11] in every region. TAO is similar to Memcached [22] or Redis [52], but provides a graph [67] API for querying vertices and edges. TAO is separated into read and write tiers, which allows specialization (DB connection management for writers, high-throughput caching for readers) and independent scaling of resources. TAO uses a read- and write-through architecture. For writes, web servers issue requests to the in-region TAO write tier. If the primary replica for the shard is in-region, the write tier directly issues a request to MySQL; otherwise, the request is proxied to the write tier in the primary region to perform the write. Web servers similarly send read requests to the local TAO read tier. On cache misses, TAO goes "upstream" to fetch the data. Figure 1 shows the upstream path – TAO first attempts to obtain the data from the local replica (1), and if unsuccessful, attempts to obtain the data from the read tier in the primary region (2) before going to the primary DB replica (3).

Keeping caches up to date. Keeping caches up to date is a known difficult problem [49]. One strategy is to use time-to-live (TTL) caching [14]. However, TTLs cause caches to either hold updated data too long (stale data) or throw away unchanged data (spurious misses). Another strategy is

to rely on read/write-through architectures to update caches on the request handling path. However, relying on a fixed topology for consistency prevents evolving the architecture for operational benefits like availability and scalability [57].

Instead, Meta uses a pub-sub system named Wormhole [56], a system similar to Apache Kafka [3] or LinkedIn's Databus [18], to stream updates to caches. Wormhole ingests MySQL's replication stream and distributes updates to subscribers such as TAO read tiers. Wormhole guarantees *at-least-once* and *in-order* delivery of updates to subscribers. While Wormhole's strong guarantees ensure the system can reliably replicate data around the world, enforcing those guarantees often leads to replication lag. For example, a hot shard or an overloaded publisher will cause lag (and data staleness) on a large number of subscribers. While average replication lag is low, Wormhole provides no bounds on maximum lag. Because both the MySQL secondaries and caches are updated asynchronously with no bound on replication lag, users are subject to *eventual consistency with unbounded staleness* [6].

Load-based subscriptions. There are multiples types of TAO tiers per region, including main and "failover" tiers. Failover tiers help improve availability and handle traffic during maintenance events. Failover tiers lazily subscribe to shard replication streams when a shard's traffic reaches a minimum threshold. These *load-based subscriptions* reduce resource usage but are frequent sources of staleness.

Write versioning. Meta's datastores use hybrid-logical clocks (HLC) [32] as a building block for consistency primitives. HLCs are versioning metadata about writes that have both a physical and logical clock component. HLCs achieve properties of both types of clocks – they capture causality between events (like Lamport clocks [35]) while also being tied to the physical time when the write was performed. HLCs can be used to detect when a data item is stale and needs to be refreshed from a more up-to-date source.

Meta's MySQL deployment mints HLCs as part of committing transactions. MySQL guarantees HLCs are monotonically increasing per shard using the formula $HLC_{new} = max(Time_{now}, HLC_{Prev} + 1)$ (where $Time_{now}$ is the host's current physical time). Meta uses NTP [42] to synchronize physical time. HLCs propagate alongside changed data through Wormhole to TAO, which caches them alongside the data. Additionally, MySQL injects a *heartbeat* [31] message (with HLC) into the replication stream every 500ms. This allows downstream subscribers to disambiguate between inactive shards and unhealthy replication – a subscriber should at least see a heartbeat every 500ms, regardless of whether there are any writes to the shard. If the subscriber does not see one, the replication stream may be unhealthy.

TAO tracks per-shard heartbeats as monotonically increasing *watermarks* indicating the shard's replication progress. Because Wormhole guarantees in-order delivery, when TAO processes a heartbeat with HLC T_j, TAO is guaranteed to have processed all updates T_i where $T_i < T_j$. In essence, TAO is guaranteed to be "up to date" as of the watermark.

2.2 Bounded staleness

Bounded staleness [40] is a read consistency level exposed by asynchronously-replicated datastores. Unlike eventual consistency, which specifies that reads to the datastore will observe a write at some point without any conditions on timeliness, bounded staleness guarantees that reads will observe any write within some system-provided time bound. Concretely, if write W_T is performed at time T and the system provides staleness bound S, then any read R performed at or after $T + S$ is guaranteed to observe W_T regardless of whether the read is to the primary or a replica. Note that under bounded staleness the reader is allowed to observe subsequent writes $W_{T+1}, W_{T+2}...$ (i.e., fresher data) but reads must reflect all older writes $W_{T-1}, ...$ that preceded W_T.

Why bounded staleness? Bounded staleness is strong enough to be useful for developers while still being scalable enough to be enabled by default for all TAO reads. Bounded staleness matches most users' mental model of the system. Enabling it by default provides highly leveraged benefits for all of Meta's developers, including reducing product raciness and eliminating debugging pain. Others have recognized bounded staleness as a powerful consistency level for users – for example, Azure Cosmos DB [41] and Spanner [23] support bounded staleness (although they target DB replicas, not the caches fronting them).

Stronger forms of consistency like linearizability [27], sequential consistency [36] and causal consistency [1] are non-starters as defaults at Meta's scale as they introduce too much communication and synchronization overhead. Previously, Meta built and deployed FlightTracker [57] to provide read-your-writes (RYW) consistency for Meta's social graph. While RYW is enabled for all TAO reads, we observed several limitations that made us turn to bounded staleness. First, users are only provided guarantees about their own writes; they still get eventual consistency for writes performed by other users or non-users like automation. Second, RYW protections are only provided by default within a region, as adding cross-region latency on every session fetch adds too much overhead to be enabled by default [57]. Finally, integrating a FlightTracker-like system in the long tail of products and services operated by Meta is time consuming and not scalable.

We have found that RYW and bounded staleness together form a compelling semantic for datastore users – a user's writes are visible to them immediately and all other writes are visible within a reasonable delay.

Picking a staleness bound. The target staleness bound must both be useful for users and be achievable considering Meta's global deployment. Setting the bound to 2 seconds struck the right compromise between usability and feasibility – it is both a useful human-scale time bound and is achievable given speed-of-light delays for cross region replication.

We evaluate Skybridge's ability to reliably provide 2 second bounded staleness in Section 5.

Bounded staleness without Skybridge. TAO in principle can implement a simple form of bounded staleness on cache hits by comparing the staleness bound S against the physical component of the watermark HLC_{wm} and cached-item HLC_{item}. When checking a cache item:

1. If $Time_{now} - (S - \varepsilon) < max(HLC_{wm}, HLC_{item})$ (where ε is max NTP clock skew[1]) then the cache item is fresh
2. Otherwise, the cache item is stale and needs to be refilled

Note that TAO uses the max of the watermark and cache item HLC. In some instances, the cache item HLC may be older than the staleness bound. This does not necessarily mean that the item is stale – it may simply have not been written recently. The replication watermark helps TAO prove these cache items as up to date.

This condition can be checked upstream all the way to the primary DB, where the data is guaranteed to be fresh. This check is local to each cache host, meaning it adds minimal overhead to reads. However, this baseline implementation causes significant operational issues. It is not fine-grained – if the shard temporarily lags beyond the staleness bound, TAO will treat *all* data on the shard as stale and go upstream to guarantee freshness (even though all data has likely not changed). This degrades TAO into a proxy rather than a cache. It can also add significant latency as requests that should be served locally may be forced to go cross-region. Finally, it creates thundering herds [37] on the primary, adding additional DB load and potentially exacerbating replication lag.

Importantly, almost all cache misses due to staleness in this implementation are false positives. Because Meta's workload is highly read-skewed, most data on a shard has not been recently written and is actually up to date. The problem is TAO does not have fine-grained information about which particular cache items have changed since the last replication watermark. What TAO needs is a *staleness oracle* to identify which cache items have been recently written. When the replication stream is lagging, TAO could check the oracle to see whether a cache item has recently changed. This would allow TAO to filter out spurious staleness misses and only refill data that is truly stale.

3 Design

Skybridge's goal is to be a fine-grained staleness oracle for TAO. When TAO receives a read request for a cache item and the replication stream backing that item is lagging, TAO consults Skybridge to determine if the item has recently changed. Skybridge responds with HLCs of writes to that item, if any; usually, however, it responds there are no recent writes. With this information, TAO can figure out whether the cache is up to date or whether it needs to refill from upstream. TAO, Wormhole and Skybridge work together to enforce 2-second bounded staleness by default for all reads.

Skybridge is complementary to the existing Wormhole replication stream, helping make bounded staleness enforcement precise and scalable. Wormhole is able to replicate most DB updates within 2 seconds. Skybridge provides the most benefit for the problematic long tail of lagging shards, where Wormhole's strong replication guarantees cause lag. Skybridge implements an independent replication pipeline and is designed to avoid the same sources of replication lag. Skybridge achieves this by relaxing its replication guarantees, which allows it to focus on real-time replication instead of durability and in-order delivery.

Skybridge has several other goals in order to support bounded staleness checking for TAO's workload. Checking against Skybridge must be low latency, as TAO cannot spend significant time doing expensive bounded staleness checks that regress request latency for clients. Additionally Skybridge must prevent as many spurious upstream fills as possible, both to avoid cross-region latency and to avoid overloading the upstream path. This means that the vast majority of bounded staleness checks must be satisfied locally on TAO hosts. Finally, Skybridge should ideally require significantly fewer resources than either TAO or Wormhole so that it can be efficiently deployed alongside existing systems.

We first describe Skybridge's replication semantics. Then, we describe Skybridge's architecture and how it implements this semantic to enforce bounded staleness.

3.1 Replication with gap detection

Skybridge implements a replication semantic called *replication with gap detection* (RGD). Systems that implement RGD have several unique properties that avoid common sources of replication lag while still enabling caches like TAO to enforce bounded staleness.

Replicating write metadata. The first property relates to who is responsible for filling stale cache entries. RGD systems only focus on identifying which cache items are stale. They leave the responsibility of refilling cache to others, e.g., TAO's upstream fills. RGD systems only need the *write metadata*, i.e., cache key (node or edge ID) and DB HLC minted by the DB, to identify stale cache entries. Note that if RGD systems wanted to instead fill cache, they would need to replicate the changed data in addition to metadata. This would require significantly more CPU, memory and network resources. Additionally, this effort would be mostly wasted – recall from Section 2.2 that most cache items are up to date even if the backing replication stream is lagging. Only a small number of reads would actually use the changed data. It is more resource efficient to instead focus only on replicating write metadata; we evaluate the resource savings in Section 5.3.

[1] If maximum clock skew exceeds ε, TAO may silently violate bounded staleness and return stale data to users. We describe how Skybridge accounts for ε in Section 4.

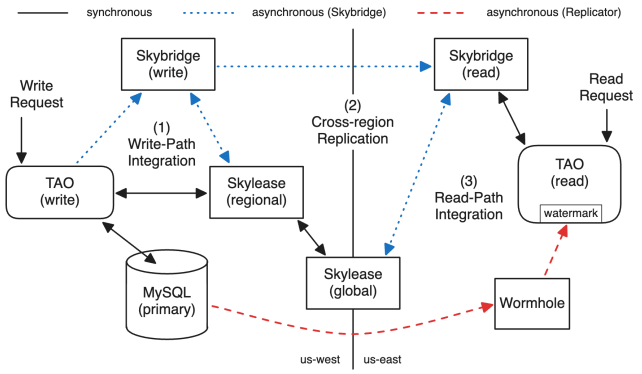


Figure 2: Cross-region replication architecture as (1) Skybridge ingests writes as they flow through TAO, (2) Skybridge replicates metadata across regions, and (3) Skybridge indexes metadata for reading from TAO.

Known data loss. The second property is that RGD allows data loss as long as it is detectable. When using an RGD system to check staleness, caches expect a "yes, write at HLC X" or "no recent writes" answer. However if the RGD system reports an "indeterminate" result (i.e., it is missing data), the cache can always conservatively assume it is stale and refill the entry. While the cache may have been unnecessarily refilled, the system as a whole still enforces bounded staleness. The system is correct as long as the RGD system can reliably detect when it is missing data. If, however, the RGD system cannot detect when it is missing data, it may falsely report "no recent writes" to the cache. The cache would silently return stale results to clients in violation of bounded staleness. Therefore the RGD system must be able to detect these "gaps" in order to provide bounded staleness. Note that this property follows directly from the first – data loss is allowable because the RGD system is not responsible for filling cache (the cache provides its own refill mechanism).

Out-of-order replication. The third property is that RGD allows out-of-order replication. This is possible because the data in the replication stream is a conflict-free replicated data type (CRDT) [55]. CRDTs are replicated data structures whose replicas are guaranteed to converge to the same state once all updates have been delivered, regardless of the delivery ordering. This requires updates be idempotent, associative and commutative. Because RGD systems implement a CRDT, they can replicate data in whatever order is most effective.

RGD replication streams are add-only set CRDTs [55] containing write metadata represented by <*key*, *HLC*> tuples. Essentially RGD systems implement a set that only ever accumulates knowledge of recent writes. The order in which write metadata is added does not matter – the set will have the same state if write A is added before write B or vice-versa. Additionally, the set will de-duplicate write metadata added more than once since the set already contains the tuple.

At read time, clients query the RGD system by *key*. If there are multiple <*key*, *HLC*> tuples in the set, the writes are merged using a last-write-wins strategy [50]. This is because the RGD system's only responsibility is to give the cache the versioning information required to produce a bounded staleness-compliant refill. Only the latest HLC matters – all previous writes will be reflected in the latest write. The RGD system returns the max of all found HLCs in order to make sure the cache observes the latest update.

Putting it all together. These three properties allow Skybridge to focus on replicating metadata about the most recent writes while de-prioritizing or even dropping older data. Skybridge's replication stream is significantly smaller than traditional replication streams and side-steps many of the traditional sources of replication lag. Because of these properties, Skybridge can focus on real-time replication of write metadata in under two seconds.

3.2 Skybridge architecture

Figure 2 shows an overview of Skybridge's architecture. Skybridge has three main components: 1) an integration with TAO's write tier to ingest write metadata about all DB writes, 2) a replication layer to fan-out write metadata to all read regions, and 3) an integration with TAO's read tier to check which cache items are stale.

Skybridge's write path (Section 3.4) ingests write metadata from TAO writers and detects gaps in the ingestion. Skybridge uses leases, which indicate when a TAO writer may write to a shard, to determine data completeness. Skybridge's client inside of TAO writers is responsible for opening leases with a system called Skylease and batching writes to send to Skybridge. Skybridge's write path receives these batches and reads lease holders from Skylease to determine when it has complete data and is therefore safe to replicate downstream.

Skybridge's replication layer (Section 3.5) is responsible for real-time replication of write metadata from the write path to the read path. Skybridge uses a pull-based model for replication – Skybridge readers pull batches of writes from the write path. This gives readers the flexibility to prioritize what data to fetch, including focusing on the newest metadata and de-prioritizing older data. The replication layer is designed to search for write metadata across multiple replicas, including write-path hosts and in-region read-path peers. If a particular window of data cannot be replicated, the replication layer will eventually give up and move on to newer data.

Skybridge's read-path (Section 3.3) indexes write metadata about all recent writes to all shards performed globally. Skybridge's indexes are designed for fast querying from TAO's read handling request path. TAO readers implement bounded staleness checks by first comparing replication watermarks to the staleness threshold (Section 2.2). If the cache item cannot be proven fresh, TAO calls into the Skybridge client for fine-grained checking. Skybridge's client sends requests to Skybridge to determine whether a cache item is stale. Addi-

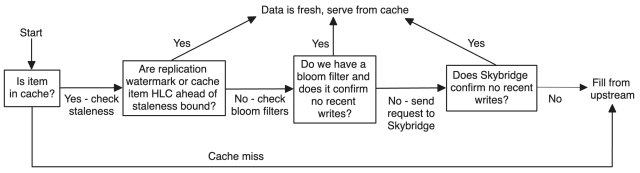


Figure 3: Request handling flow for TAO reads

tionally the client preloads Skybridge's indexes on the TAO host for lagging shards to minimize the number of required external requests. If Skybridge reports the item is stale or if Skybridge has missing data, TAO goes upstream to refill cache. Otherwise TAO serves the cache to clients.

3.3 Read path

TAO interacts with Skybridge during handling of read requests. The main goal is to determine whether the item that TAO has in cache is fresh with respect to the 2-second staleness bound. TAO achieves this through a combination of Wormhole watermarks, Skybridge requests and local caching of Skybridge metadata. This checking is designed to be precise and lightweight, limiting bounded staleness cache misses only to those cache items that have actually recently changed and without adding significant cache request latency.

Figure 3 shows TAO's bounded staleness control flow. First, TAO checks whether the item is in cache; if it is a cache miss, bounded staleness checking is not required. Next, TAO constructs a cache HLC, HLC_{cache} from the max of the watermark and cache item HLC (Section 2.2). If the physical component of this HLC is newer than the staleness bound, the item is fresh and TAO returns it to the caller. If not, TAO consults with Skybridge to determine whether the item was written between $[HLC_{cache}, StalenessBound]$, where $StalenessBound = Time_{now} - (2\ seconds - \varepsilon)$. This interval is referred to as the *lag interval*. Listing 1 shows the two Skybridge APIs that TAO uses to check whether there were any recent writes to the cache item within the lag interval.

Querying recent writes. Skybridge's main API, `getWrites`, takes a cache item key and lag interval as arguments and returns the latest HLC of all DB writes to that key within that interval (TAO also supports range scans [47], for which Skybridge returns the latest HLC for all keys in the range). `getWrites` also indicates whether it had complete data for the interval (discussed further below). If Skybridge's indexes were complete then TAO compares the cache HLC to the list of recent HLCs. If there are no recent writes or if the cache HLC is greater than any HLC in the list, then TAO can serve the data. If Skybridge returns an HLC higher than the cache HLC, TAO sends an HLC-conditional request upstream with that HLC; any upstream host that satisfies the HLC can fill the data. Finally, if Skybridge's indexes are incomplete or there was an error talking to Skybridge, TAO refills from upstream using *StalenessBound* as the condition. Note that Skybridge is only checked by the first cache host – other upstream hosts simply forward the request upstream if their cache HLC is too old. It adds too much latency to re-check Skybridge on every hop.

Bloom filters. TAO can only send `getWrites` requests to Skybridge after calculating cache HLCs, meaning Skybridge requests are on the cache read request critical path. However, TAO can actually predict when it will have to start checking staleness with Skybridge by monitoring replication stream lag. If TAO can proactively load Skybridge's indexes for lagging shards, it can perform staleness checks locally rather than sending requests to Skybridge. The main downside of this approach is that even though Skybridge's indexes are small, they are still large enough to compete with TAO's caches for memory. Storing Skybridge's indexes locally could regress TAO's hit rate and cache request latency.

Instead, Skybridge uses bloom filters [58] built from its indexes. Bloom filters can have false positives, i.e., the bloom filter may indicate the data has changed even when it actually has not. The bloom filters do not, however, have false negatives. This allows TAO to correctly enforce bounded staleness. The set of recent writes on a shard is sparse, meaning the generated bloom filters are minimal in size. TAO pre-loads bloom filters for lagging shards, helping prove a significant portion of read requests up to date without having to send a request to Skybridge. This also significantly reduces the request load on Skybridge. Note that Skybridge only sends bloom filters for complete windows; there is no reason to send them for incomplete windows since TAO would have to consult Skybridge or go upstream anyway.

Listing 1 shows Skybridge's bloom filter streaming API. Skybridge chunks its indexes into discrete intervals and builds bloom filters that are continuously streamed to clients. TAO monitors replication lag across all shards and opens a bloom filter stream for a shard when its replication watermark is older than 1.5 seconds. This allows TAO to pre-load bloom filters for checking if lag increases beyond 2 seconds. When the replication stream's lag drops below 1.5 seconds, TAO closes the bloom filter stream.

Similarly to `getTicket` requests, TAO checks local bloom filters for writes within the lag interval. If TAO has bloom filters covering the entire lag interval and they indicate no recent writes, TAO can prove it has fresh data and serve it from cache. Otherwise, TAO sends a `getTicket` request to Skybridge to get a definitive answer.

Skybridge server-side indexes. Skybridge's indexes are sharded by DB shard and store *write windows*. A write window is composed of a lower HLC bound, an upper HLC bound and a list of write metadata for all writes to that shard performed in that write window. Skybridge hosts replicate metadata from DB primary regions, where TAO hosts publish metadata about writes they performed to the local Skybridge tier. `getTicket` requests scan write windows overlapping the lag interval to search for any writes to the specified cache key.

```
struct Interval{i64 lower; i64 upper;};
struct Response{
  list<tuple<string, i64>> hlcs; bool complete;};
struct BloomFilter{
  Interval interval; binary serializedFilter;};
// Read recent writes from Skybridge
Response getWrites(string key, Interval lag);
// Stream bloom filters from Skybridge
stream<BloomFilter> openBloomFilterStream(
  i64 dbShard, i64 start);
```

Listing 1: Skybridge read API. TAO can request recent writes for a key and interval or stream serialized bloom filters constructed from Skybridge's indexes.

When responding to queries, Skybridge needs to determine whether it has complete knowledge of all writes to a shard (i.e., no gaps). Given a lag interval I_{lag}, index interval I_{index} (formed by the union of all write windows indexed by Skybridge), and write window D, Skybridge has complete data if:

1. $I_{lag} \subseteq I_{index}$ (the lag interval is covered by write windows indexed by Skybridge)
2. For all $D \in I_{lag} \cap I_{index}, D.complete = true$ (all indexed windows covering the lag interval are *complete*)

If both criteria are satisfied, Skybridge has complete information about every write to the shard for the lag interval and can provide definitive cache freshness answers to TAO. Section 3.4 describes how Skybridge's write-path integration determines whether write windows are complete and Section 3.5 describes how Skybridge drives replication of write windows from primary regions for every DB shard.

3.4 Write path

Skybridge's goal on the write path is to ingest metadata about all writes committed to the DB. Skybridge ingests write metadata by collecting *heartbeats* from TAO write hosts. TAO writer heartbeats are defined by a lower and upper HLC bound and a list of writes committed by that TAO writer within those bounds. TAO writers guarantee that heartbeats sent to Skybridge contain all writes to the DB performed by that writer during that interval. Skybridge hosts aggregate heartbeats from TAO writers into write windows to publish downstream.

Skybridge uses a leasing system called Skylease to track which hosts are writing to which shards. Any single TAO writer only serves traffic for a small fraction of DB shards. Leases [24], which indicate the time period when a writer may be writing to a shard, make heartbeating efficient by only requiring TAO writers to send heartbeats for shards they own (i.e., avoid sending heartbeats from the cross-product of TAO writers and DB shards). Skybridge's client is responsible for opening leases on hosted shards and batching write metadata into heartbeats to send to Skybridge. Skybridge must collect heartbeats from all of a shard's lease holders, i.e., writers, in order to have complete data for that shard.

Leases. Leases are a contract between TAO writers and Skybridge. A writer is only allowed to write to a shard if it has an open lease and the host promises to send heartbeats for the duration of the lease. If for whatever reason a lease holder fails to send a heartbeat (e.g., host crash), Skybridge will detect data loss by observing a gap in expected heartbeats. This is the core of how Skybridge implements replication with gap detection's known data loss guarantee.

Leases are denoted by a *lease holder* (e.g., TAO host), *lease object* (e.g., DB shard) and HLC interval. Leases are non-exclusive since multiple TAO writers may be handling writes for a given DB shard for higher availability. Lease HLC ranges are minted by Skylease hosts based on their physical clock and the lease duration requested by the client, i.e., $[Now, Now + Duration)$. We describe why using physical clocks does not violate correctness below.

Skylease must be 1) highly available to avoid blocking write handling, 2) durable to avoid losing leases and causing unknown data loss, 3) strongly consistent so that Skybridge knows the exact set of lease holders at a given time, and 4) scalable to leases on millions of lease objects. To satisfy these requirements, Skylease acts as a batching and sequencing control plane on top of strongly-consistent distributed consensus systems like Apache Zookeeper [60] or Delos [8].

Skylease employs a number of techniques to scale heavyweight consensus rounds to millions of leases. First, Skylease's protocol uses delta compression [30]. Because a writer's shards do not frequently change, Skylease's protocol only records changes between subsequent open lease requests. Second, Skylease scales out by using its own sharding, where lease objects are hashed to a Skylease shard and each Skylease shard is backed by a consensus group. Finally, Skylease batches writes to a Skylease shard from multiple writers to reduce the number of consensus rounds.

Skybridge must handle the situation where reading lease holders races with writers opening leases. For example, if a TAO writer is allowed to open a lease on a shard after Skybridge has read the lease holders, Skybridge could experience unknown data loss by missing a lease holder and therefore missing heartbeats. To prevent this situation, Skylease periodically *seals* leases on a shard by bumping a monotonically increasing watermark as part of consensus rounds. This watermark represents the lowest valid HLC of any newly opened leases. If a TAO writer tries to mint a new lease with an interval below or overlapping the watermark, Skylease rejects the lease by sequencing the operation through storage. Skylease also responds to Skybridge queries (e.g., "read lease holders for shard S between $[T_x, T_y)$") with "not sealed" until the interval is completely sealed. This ensures Skybridge sees the full lease holder set when building write windows.

Skylease maintains an in-memory view of lease holders, including materializing the current watermark and full set of TAO leases from delta compressed storage. Read queries against this view are fast enough to allow frequent polling by

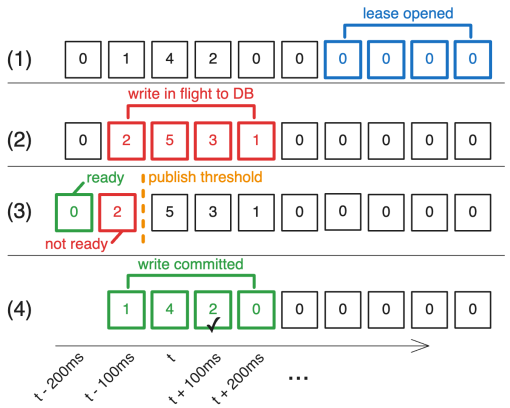


Figure 4: Skybridge's client maintains heartbeats (shown as boxes) sorted from old to new with counts of in-flight writes. (1) Leases opened with Skylease are divided into heartbeats and added to storage. (2) Before writing to the DB, TAO acquires and creates HLC bounds from heartbeats. (3) The client asynchronously scans for heartbeats to publish that are older than a threshold and do not have any in-flight writes. (4) After the write is committed, TAO releases heartbeats and stores the write in the appropriate heartbeat (check mark).

Skybridge until the queried interval is sealed. Skylease maintains watermarks per Skylease shard; watermarks apply to all lease objects on the shard. When advancing the seal watermark, Skylease keeps the new watermark behind the current physical time to avoid impacting open lease availability.

Building heartbeats. Leases ensure Skybridge is able to determine which writers should be sending heartbeats for which shards. TAO writers are responsible for sending complete heartbeats for the entire opened lease duration to Skybridge. A heartbeat H is *complete* if all writes W committed by the writer between the heartbeat's lower and upper HLC bounds are contained in the heartbeat's write set:

For all W such that $H_{lower} \leq W.hlc < H_{upper}, W \subseteq H.writes$

Note that this implies that once a heartbeat is sent to Skybridge, no more writes may be committed within that heartbeat's bounds so as to avoid violating the completeness property for the sent heartbeat.

TAO writers must only write to the DB within valid lease bounds to ensure that Skybridge correctly tracks all writers. To achieve this, TAO writers call into the Skybridge client before writing to the DB in order to read the current lease or block until one is acquired. TAO then attaches the lease HLC bounds to the DB request. Meta's MySQL deployment allows specifying HLC bounds on transactions – if the HLC minted by the DB for a transaction does not fall within the bounds, MySQL will abort the transaction. This ensures MySQL respects the leases opened by TAO writers. This HLC coordination with the DB is why it is okay to mint leases based on Skylease host physical times - the HLC bounds will ensure the write falls into the appropriate time range.

In order to build complete heartbeats, Skybridge's client 1) maintains HLC bounds for DB write requests to avoid minting new write HLCs outside of leases or already-sent heartbeats and 2) tracks outstanding writes (including in which heartbeats a write may land). The Skybridge write client achieves both goals using an *in flight write* storage for tracking heartbeats and writes. This data structure maintains a list of heartbeats to be sent to Skybridge for each DB shard. Each heartbeat in storage tracks how many in-progress writes that could land within that heartbeat's bounds. This storage ensures heartbeats are complete before being sent to Skybridge.

Figure 4 shows how the client uses this storage. (1) shows heartbeats getting added to storage based on leases opened with Skylease. Each lease gets divided into smaller discrete heartbeats since lease durations are much longer than individual heartbeats. The heartbeats in storage are what TAO promised to send to Skybridge but has not yet sent. Any interval outside of storage either does not correspond to an open lease or represents a heartbeat that has already been sent to Skybridge. In other words, future writes to be committed by the DB must land in one of the heartbeats in storage.

(2) shows a write request coordinating with the storage before going to the DB. The client generates HLC bounds for the transaction based on heartbeats in storage. To track outstanding writes, the client *acquires* heartbeats in which the write could land. This operation (maintained as a reference count on the heartbeat) blocks the heartbeat from being published until the write has been committed by the DB. (4) shows the post-DB processing – all heartbeats acquired in (2) are *released* and the completed write's metadata is stored in the heartbeat containing its HLC. If there was an error writing to the DB, the heartbeats are simply released without storing any metadata. While the client could acquire every single heartbeat in storage, most writes have HLCs that fall within a narrow range of heartbeats near the current physical time. The client is configured to only acquire a small subset of heartbeats, which also helps prevent straggler writes from blocking large numbers of heartbeats from being published.

(3) shows how the client determines when heartbeats are ready to be published to Skybridge. The client periodically scans all shards in storage looking for heartbeats that are older than a publish threshold (which lags the host's current physical time) and that do not have any outstanding writes. If there are still writes in flight that may land in a particular heartbeat, the client skips it. Otherwise the heartbeat is dequeued from storage and sent to Skybridge. At this point the heartbeat is sealed and all future writes will be prevented from landing in it through the HLC bounds created in (2). Note that the client takes advantage of out-of-order publishing to avoid delaying the sending of heartbeats. The client can send any heartbeat once it is ready for publishing, even if there are older heartbeats blocked due to straggler writes.

Tuning lease and HLC bound sizes. Lease durations and HLC bounds represent a trade-off between Skylease load and "blast radius" for when things go wrong. Lease duration should ideally be as low as possible, but is bounded by how quickly Skylease can run consensus rounds. Alternatively, TAO could open very long leases, e.g., on the order of minutes, to minimize Skylease load. However if the TAO host crashed shortly after opening a long lease, Skybridge would have minutes of data incompleteness. Similarly, for heartbeats acquired before going to the DB, the bounds should be as small as possible to minimize the number of heartbeats blocked on stragglers, but large enough to give the DB ample time to process the write (too small HLC bounds will cause spurious aborts). The Skybridge client is tuned to open leases on the order of tens of seconds and acquire HLC bounds on the order of hundreds of milliseconds.

Aggregating heartbeats. Skybridge aggregates heartbeats from in-region writers into write windows for a shard by 1) polling Skylease for the (sealed) list of all lease holders, and 2) waiting for heartbeats from TAO writers covering those leases. Skybridge knows it has a complete write window D if for Skylease watermark $S_{watermark}$ and heartbeats H corresponding to lease L:

1. $D_{upper} < S_{watermark}$ (Skylease has sealed the lease holder set past the write window's bounds)
2. For all L such that $D_{lower} \leq L_{lower} < L_{upper} \leq D_{upper}, H_L \subseteq D$ (Skybridge has all heartbeats from all lease holders for the write window interval)

Once Skybridge has all heartbeats from all lease holders for a given shard and interval, it builds a complete write window ready for publishing. Skybridge is configured with a timeout on how long it will wait for heartbeats. If the timeout expires and it still has not received all heartbeats, Skybridge will publish an incomplete write window to give subscribers some information about recent writes. Skybridge will, however, continue waiting and if the heartbeats later arrive it will republish the complete window downstream. Skybridge again takes advantage of out-of-order replication to avoid blocking complete windows from replication if an older window is waiting for data from TAO.

3.5 Cross-region replication

Skybridge's replication layer is responsible for replicating write windows from all primary regions to regions serving read traffic within 2 seconds. While Skybridge's correctness criteria allows gaps, Skybridge tries to replicate complete data as frequently as possible to be effective on TAO's read path. Because of the real-time nature of replication, Skybridge's replication layer fetches missing data out of order and from multiple replicas using a publisher-subscriber [3] model.

While TAO writers push windows to the local Skybridge tier, cross-region subscribers pull a continuous stream of write windows from publishers to fill their indexes. Using a pull model gives subscribers the ability to quickly detect gaps and re-fetch windows from another source. Because write windows are conflict-free replicated data types, subscribers have a lot of flexibility to reorder and retry data fetches. Using a push model would require publishers to discover missing data from subscribers, requiring too much coordination.

The replication layer builds a continuous stream of write windows with newer windows sent first. Early versions of the replication layer opened a single long-lived stream per shard using a protocol similar to gRPC streams [25]. However Skybridge frequently experienced head-of-line blocking and could not prioritize newer data while the stream tried to flush out older windows. Additionally, long-running streams caused load imbalance over time as Skybridge deployments rotated hosts through maintenance. Long-lived publishers accumulated more streams and eventually became overloaded until they were restarted (and streams moved to other hosts).

Instead, Skybridge's replication layer opens short seconds-long streams. This gives more load balancing opportunities across publishers and allows subscribers to de-prioritize older sections of the stream by either delaying re-fetching them or not re-fetching at all. Subscribers periodically open new streams to primary regions to stream the most recent data. Publishers may send write windows out-of-order, publish duplicate or incomplete windows or even leave gaps. None of these block subscribers from continuing to stream newer data. However when a subscriber detects missing data (e.g., an incomplete write window), it schedules another attempt to stream the data after some delay. This helps when heartbeats from TAO are delayed, e.g., blocked waiting on a DB write. Additionally this allows subscribers to try re-fetching from other sources. Each primary region contains two Skybridge replicas, meaning there are two possible sources of writes. In-region replicas ("peers") will also exchange write windows, which helps quickly warm up Skybridge hosts after a restart.

Finding primary regions. Subscribers need to know each shard's primary region in order to fetch that shard's writes. Similarly to TAO's write path, Skybridge uses leases at the global level to track each shard's primary region. For Skybridge to guarantee completeness, regional leases must only be opened within a global lease and therefore the region must have a global lease before any regional leases can be opened. When handling an open lease request from TAO, regional Skylease hosts implicitly open global leases for the DB shard and region with Global Skylease. Global leases are typically opened for much longer intervals (e.g., on the order of minutes) as shards rarely move between regions.

Subscribers drive replication for a shard by reading global leases from Global Skylease. The replication layer uses the region name (i.e., lease holder) and lease bounds to stream data from the appropriate primary region.

4 Implementation

Skybridge provides a means for TAO to enforce bounded staleness on read requests. Below we discuss a number of practical considerations regarding TAO and Skybridge.

Circuit breakers. Although TAO aims to provide bounded staleness for all requests, Skybridge cannot regress TAO's availability and performance. On the write path, too many writes blocked waiting for leases from Skylease will trip a circuit breaker and TAO will let writes proceed without leases. Similarly, too many read requests trying to go through the high-overhead portion of bounded staleness checking (requests to Skybridge, consistency misses upstream) will trip a circuit breaker and TAO will serve whatever is in cache. These circuit breakers prevent cascading failure modes. For example, if there is widespread Wormhole lag, bounded staleness checking can create a thundering herd on the upstream path towards the DB which could make lag worse.

TAO "fails open" when tripping circuit breakers. TAO fires an alarm whenever these rate limits are hit to notify Meta engineers about issues that need investigation. In fact these alerts have notified us about Wormhole and TAO problems before either of the teams knew there was an issue.

Fail-closed vs. fail-open While TAO defaults to failing open if it hits rate limits, sometimes product developers want stronger consistency at the expense of availability – they would prefer the request failed if TAO cannot guarantee bounded staleness. TAO provides a "fail-closed" API for this scenario. Clients opt in to the API by setting a flag on the TAO request that tells TAO to fail the request if it cannot guarantee bounded staleness. TAO reserves a portion of its rate limit budget for fail-closed requests to prioritize their availability.

Extended retention Skybridge is an in-memory service and is memory-capacity bound (see Section 5.3). Skybridge adjusts retention of its indexes to keep memory usage stable as write traffic fluctuates over time. Skybridge maintains a baseline retention for all DB shards, but TAO shard lag is not uniform – individual shards may have worse lag than others. Skybridge can selectively extend retention for a small set of these longer-lagging shards. Skybridge detects these shards based on TAO queries – if it sees multiple requests for a shard where the lag interval is approaching retention limits, Skybridge marks the shard as "extended" and maintains write windows for that shard for a significantly longer interval.

Clock skew Meta's hosts use NTP [42] to synchronize physical clocks. NTP is known to have on the order of hundreds of milliseconds of clock skew. Meta is investigating using more precise clock synchronization such as PTP [46, 54], but for now we move the staleness bound forward by 50ms to try to account for clock skew.

Emulating primary-only and causal reads TAO provides linearizable reads that bypass cache and read from the primary DB. To avoid DB overload issues, TAO actively discourages using them unless absolutely necessary. With fail-closed bounded staleness reads, however, TAO has a lightweight drop-in replacement. When clients call this replacement API, the web server sleeps for 2 seconds to wait out the staleness bound and then sends a fail-closed request to TAO. TAO either guarantees consistency or returns an error to the client if it cannot (e.g., hit upstream rate limits or network errors). This means clients get linearizable reads. Similarly, bounded staleness can provide a limited form of causal reads. If the client knows when a write occurred (i.e., HLC of the write) then it can wait until 2 seconds have passed to perform the read. TAO and Skybridge will ensure the write is visible. These APIs are highly efficient from TAO's perspective since the client waits for replication and TAO likely serves data from cache.

Clients can also use these APIs as a stronger backup when regular reads are not able to see a relevant write. Clients first issue a regular read, which succeeds most of the time. On the infrequent occasion that TAO is stale (e.g., clients do not see an object they created), they wait for 2 seconds then retry the read with stronger consistency. Bounded staleness enables these new APIs without any external synchronization or forcing clients to pass around HLCs.

Failure scenarios and fault tolerance. Like most distributed systems, Skybridge must contend with single-node failures. However, because of replication with gap detection semantics, individual Skybridge host failures do not cascade into replication lag or other outages. Skybridge's replication layer is resilient to these issues. There are multiple Skybridge write-path hosts per DB replica from which downstream subscribers can stream updates. Similarly, there are multiple Skybridge read-path hosts per DB replica from which TAO readers can open bloom filter streams and query for recent writes. Skybridge's replication layer is free to stream updates from multiple sources in whatever order is best.

While network partitions between TAO and Skybridge are technically feasible, we have not noticed them. Region drains and cuts are more likely in practice, especially as part of routine testing [10]. However, Skybridge's leasing system gracefully works around them. Recall that TAO writers (and Skybridge write-path hosts) are co-located in the same region as primary DB replicas. During a region cut, the region no longer receives any write requests. TAO writers no longer open leases and subscribers stop attempting to subscribe to updates from that region soon after. Therefore Skybridge, TAO and MySQL all share the same fate in a region outage and Skybridge in other regions still functions correctly.

Skybridge's read path has showcased several priority inversion [4] failures. A priority inversion exists when a higher priority task is impacted by a lower priority task. For Skybridge's read path, this manifests as read traffic from TAO affecting Skybridge's replication health. This typically occurs when Skybridge hosts have temporary spikes of incomplete data (e.g., a TAO writer crashing leading to a gap). In this scenario, Skybridge cannot send bloom filters to TAO since

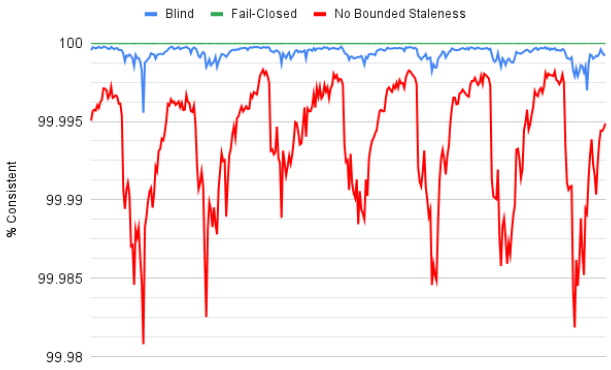


Figure 5: Bounded staleness consistency over 7 days with no bounded staleness (red), blind (blue) and fail-closed (green) reads. TAO's 2-second consistency without Skybridge is 99.993% but frequently drops below 99.985%. Skybridge significantly increases consistency to 99.9993% for blind and 99.99998% for fail-closed reads.

it has incomplete data. This causes TAO hosts to fall back to `getWrites` queries to Skybridge. `getWrites` queries are more expensive than publishing bloom filters, especially since bloom filters help TAO elide further `getWrites` calls. The increase in `getWrites` traffic starves hosts of processing resources to drive replication, leading to more data incompleteness, fewer bloom filters and more `getWrites` traffic. Skybridge gets stuck in a metastable failure state [29] in which the service is degraded until remediated by outside intervention. To avoid this cycle, we added a priority queue for threads in Skybridge hosts to prioritize replication, then bloom filter streams and finally `getWrites` traffic. This ensures replication and bloom filter streams are healthy, minimizing the number of `getWrites` calls from TAO.

Similarly, Skybridge write-path hosts also experienced priority inversion. Skybridge write-path hosts ingest heartbeats from TAO writers and publish write windows downstream to subscribers. As mentioned in Section 3.5, the replication layer attempts to re-fetch missing windows. Many downstream subscribers re-fetching data can compete with TAO writer heartbeats, causing Skybridge write-path hosts to drop heartbeats. This causes data loss, leading to more subscribers attempting to re-fetch and further competition for resources. Skybridge again solves this problem using a priority queue to ensure heartbeat ingestion takes precedence over downstream subscribers. Additionally, Skybridge write-path hosts rate limit the number of subscribers re-fetching windows.

Skylease demonstrated another interesting failure mode. When Skylease is unhealthy and cannot provide leasing functionality, TAO writers fail-open on write requests and handle writes without leases. Additionally, Skybridge is not able to read lease holders (since Skylease cannot seal shards), leading to Skybridge publishing all of its write windows as incomplete. Skylease can become overloaded when the mapping of DB shards to TAO writers rapidly changes. For example, in one instance a buggy TAO release caused write requests to get misrouted to TAO writers. Because TAO writers handle misrouted requests to maintain availability, they opened leases for the misrouted requests as well. Skylease's delta compression was ineffective in this situation, leading to a spike in throughput to Skylease's storage backend and out-of-memory crashes when materializing the set of lease holders. This caused excessive downstream incompleteness, leading to a large number of fail-open reads and writes. To prevent this situation, we added rate limits on how many lease changes can be sent per request. While this means there occasionally may be a brief spike in fail-open writes, it ensures Skylease is still able to seal shards and Skybridge is able to publish write windows downstream in a timely manner.

5 Evaluation

In this section we evaluate Skybridge's ability to reliably and efficiently provide bounded staleness for TAO.

Experimental setup. TAO runs in tens of regions around the world, handling millions of write queries and billions of read queries per second. For Skybridge to be effective on TAO's read path, it must be able to improve staleness for recently written data while filtering out almost all spurious staleness misses that would cause TAO to go upstream (and likely out of region). Below we describe Skybridge's effectiveness for Meta's production workload.

5.1 2-second staleness improvements

We built a staleness checking service (referred to as the *checker*) to monitor 2-second consistency. TAO writers send a sample of writes to an in-region checker instance. On receiving a write, the checker sleeps for 2 seconds from the write's HLC and then issues read requests for that write to all TAO tiers in all regions. The checker sends three types of requests:

- **No bounded staleness.** The checker explicitly disables bounded staleness protections. This is what users experience without Skybridge.
- **Blind.** The checker issues a normal read request. TAO implicitly applies bounded staleness protections but can fail open if it hits rate limits.
- **Fail-closed.** The checker issues a fail-closed request – TAO returns an error if it cannot guarantee bounded staleness.

Figure 5 shows the difference in 2-second consistency observed by the checker over a 7 day period. By default, TAO provides 99.993% 2-second consistency relying solely on Wormhole and frequently drops below 99.985%. With best-effort Skybridge, however, consistency jumps to 99.9993%, reducing the number of inconsistent reads without any user intervention. The main limiting factor preventing better consistency is TAO's bounded staleness rate limits, which limit the number of Skybridge and upstream queries and causes

Fail-open reason	% of fail-open requests
Long shard lag	95.9%
No replication stream	3.3%
Skybridge replication lag	0.3%
Missing data from write path	0.2%
Skybridge request error	0.2%

Table 1: Fail-open reasons by % of all fail-open requests

the read to fail open. When the checker opts into fail-closed mode however, the consistency rate climbs to 99.99998%. This means only a couple thousand out of billions of requests are stale after 2 seconds. At this scale, clock skew begins to make a difference (recall staleness thresholds are created by a host's physical time). Additionally the checker does not have a way to opt the associated writes into fail-closed behavior since it only observes writes after the fact, meaning it may be checking writes that are not tracked by leases. Regardless, this number has proven to be high enough to be used by products that require strong consistency guarantees for correctness.

We further investigated the reasons for inconsistency in the blind/fail open case. Table 1 shows fail-open reasons as reported by TAO. The biggest cause of inconsistency was long shard lag. Because Skybridge is an in-memory service, it only maintains the most recent writes (see Section 5.3 below). Skybridge's retention cannot cover shards that are lagging beyond several minutes, meaning TAO must fall back to going upstream. TAO hits upstream rate limits when a significant number of shards lag by more than several minutes.

The next biggest cause of inconsistency is reads to shards without a replication stream, i.e., shards whose load was too low to trigger a subscription (Section 2). Without a subscription, there is no replication watermark and the TAO host cannot check staleness (including using Skybridge) for that shard. The host must fall back to going upstream. Traffic on TAO failover tiers frequently hit these rate limits.

The remaining causes of inconsistency are lag in Skybridge's replication layer, missing heartbeats from TAO's write path and errors talking to Skybridge. Missing heartbeats from TAO writers is outside of Skybridge's control as windows cannot be sent to Skybridge until all outstanding writes are acknowledged by the DB. We are continuing to optimize the replication layer and Skybridge's request handling to address the other two sources of inconsistency, although they comprise only a small fraction of fail-open requests.

TAO's rate limits are tuned to avoid overloading the upstream path and regressing serving latency. Sending too many requests upstream can regress aggregate latency and cause an increase in timeouts for products. Because we wanted bounded staleness enabled by default, we tightened the rate limits to minimize user impact; the current limits strike a good balance between strong consistency without user-visible performance impact.

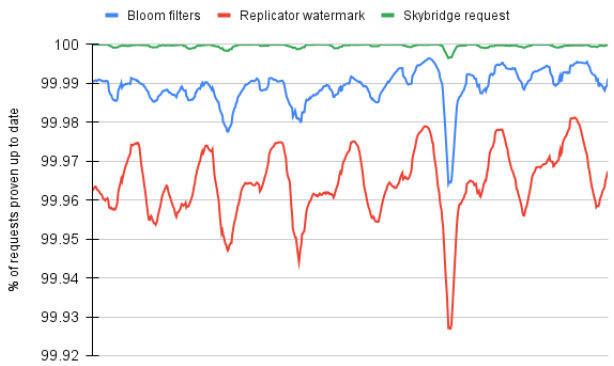


Figure 6: Percent of read queries proven up to date using the Wormhole watermark (red), host-local bloom filters (blue) and an in-region request to Skybridge (green)

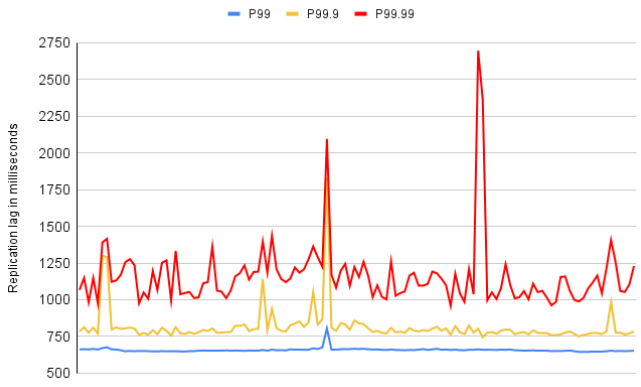
5.2 Traffic analysis

Skybridge helps TAO provide bounded staleness for all read queries because it dramatically reduces the number of queries that must be sent upstream to guarantee freshness. We evaluated how frequently Skybridge is involved in TAO's read handling and how effective Skybridge is in helping filter out upstream requests. Figure 6 shows the percentage of read requests proven up to date solely based on the watermark (i.e., the Wormhole replication stream is not lagging), percent of up-to-date requests with the addition of Skybridge's eagerly-streamed bloom filters (i.e., the bloom filter says there are no recent changes), and percent of up-to-date requests with an authoritative request to Skybridge. Note that these numbers exclude requests to shards without a replication subscription, i.e., load-based subscription.

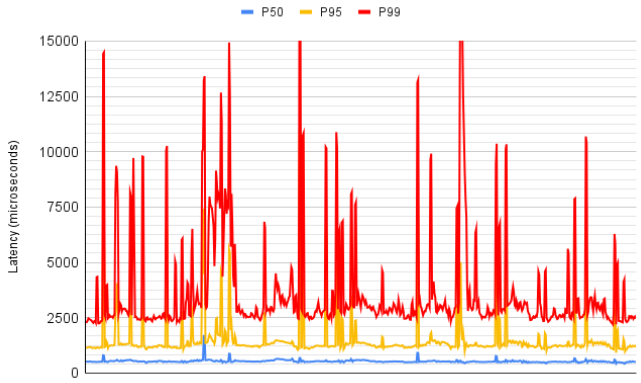
Using Wormhole watermarks allows TAO to prove 99.96% of reads up to date. Checking against host-local bloom filters increases that to 99.98%, meaning tens of millions of additional read queries per second can be proven up to date. Because bloom filters are in-memory on the TAO host, they incur almost no additional latency on the request handling critical path. Including requests to Skybridge, TAO is able to prove 99.9996% of requests up to date. While a request requires a network hop, it is low latency since it is an in-region request. Figure 7b shows P50, P95 and P99 request latency to Skybridge. Requests take on the order of milliseconds, meaning TAO can quickly get a staleness answer from Skybridge. Overall, Skybridge is able to provide staleness bound checking with low-overhead, with only 0.0004% of requests needing to go upstream to get fresh data.

5.3 Replication and footprint

Figure 7a shows Skybridge's P99, P99.9 and P99.99 replication lag, defined as the difference between the Skybridge host's physical time and the upper bound of the replicated



(a) Skybridge replication lag at P99, P99.9 and P99.99



(b) Skybridge latency at P50, P95 and P99 percentiles

Figure 7: Skybridge replication and query performance

write window. P99 replication lag is stable at around 700ms. Even at P99.99, Skybridge is able to replicate windows around the world within 1.5 seconds outside a few spikes. This real-time delivery is a combination of small payload size and out-of-order replication. Delivering 99% of the data in ∼700ms opens up possibilities for sub-2-second staleness bounds. For example, the primary read emulation API could offer tighter latency bounds for use cases that cannot tolerate waiting for 2 seconds, although there may need to be a cap on max throughput to avoid hitting rate limits and triggering errors.

Skybridge requires a relatively small number of servers. The entire Skybridge ecosystem (Skybridge read and write path, regional and global Skylease) uses 0.54% of TAO's server footprint. Skybridge is able to retain between 93 and 109 seconds' worth of recent writes as write traffic fluctuates throughout the day (Skybridge is memory-capacity bound). A single Skybridge tier pulls information about all committed writes from all primary regions, requiring between 4.8-7.9GB/s of network bandwidth. Wormhole's replication stream to TAO, meanwhile, requires between 170-300GB/s. These numbers demonstrate that Skybridge is lightweight compared to TAO and Wormhole.

RGD semantics allow Skybridge to be minimal in size while still providing staleness information about recent writes in real time. Skybridge takes full advantage of its core principles (only replicate metadata, known data loss, out-of-order replication, decoupling from Wormhole) to drastically boost TAO's 2-second bounded staleness up to 99.99998%. It is able to achieve these benefits while helping TAO avoid going upstream for 99.9996% of queries. Skybridge is a reliable and efficient means for boosting TAO's user experience.

6 Limitations and future work

Increased retention. Skybridge's indexes are memory capacity bound and only cover several minutes' worth of replication lag. Increasing the number of hosts or memory capacity per host would boost retention. Another strategy would be to spill older sections of indexes onto disk (e.g., flash drives). This would allow Skybridge to expand retention into the tens of minutes range but would require care to avoid regressing query latency. Other systems like Memcached [20] are able to support disk storage with minimal regressions, meaning it is likely a feasible solution for Skybridge.

Dynamic sharding. TAO statically assigns data to shards, but systems like Cockroach DB [33], BigTable [12], Spanner [16] and TiDB [28] are able to dynamically reshard data to balance load. Skybridge's leasing mechanism could be extended to handle dynamic resharding, but would likely require integration with the resharding control plane to safely coordinate changes.

Global secondary indices. Meta uses global secondary indices [9] to materialize queries that would require many TAO requests, e.g., inverted indexes on fields. A shuffle service ingests updates from Wormhole and generates updates for global secondary index leaf hosts. Skybridge would need to tackle several problems to provide bounded staleness for these indices. First, indexing systems need a way to track a replication watermark across the large fan-in (potentially millions) of input DB shards. Second, indexing systems must have a way to "repair" a given index when determined to be missing some writes (similarly to TAO filling from upstream). We leave both of these problems as future work.

7 Related Work

Strong consistency on eventually-consistent datastores. There are many works that augment eventually-consistent datastores with stronger consistency. Riak [53] and Cassandra [34] provide opt-in linearizability through quorum reads of replicas. Gemini [38] uses logical clocks to direct cross-replica reads for serializability. RAMP [5] (and Facebook's RAMP-TAO [13] variant) provides read atomicity for transactions. All of these these systems require opting in to heavyweight protocols, meaning only a small fraction of requests see improved consistency. Skybridge instead focuses on enabling bounded staleness by default for all reads.

Bounded staleness in datastores. There are a number of datastores that implement bounded staleness. Azure Cosmos DB [41] and Google Spanner [23] implement bounded staleness in multi-region setups. Both datastores provide bounded staleness as an opt-in mechanism and backpressure (i.e., throttle or block) if the local replica is stale. Skybridge instead provides on-by-default bounded staleness for distributed caches, which significantly more traffic.

C5 [26] and CRAQ [59] propose new replication and concurrency protocols for ensuring DB replicas are kept up to date and provide strong consistency. Query Fresh [61], Harmonia [68] and IONIA [62] propose using new networking and storage technologies to either prevent lag or eliminate conflicts between replicas. None of these works address lag in large-scale pub-sub systems such as faulty publishers or network congestion like what is present in Meta's serving stack (and what causes replication lag for TAO).

PolarDB-SCC [64], similarly to Skybridge, uses granular staleness information to allow serving consistent reads from replicas. The main difference between PolarDB-SCC and Skybridge are the target workloads – PolarDB-SCC targets strong consistency for a single DB replicated across multiple machines, whereas Skybridge targets bounded staleness for millions of DBs spread around the world. There are also implementation differences such as deriving staleness bounds by reading from the primary vs. using local physical time and waiting for the replica to become up to date vs. refilling from upstream. Nevertheless, both approaches optimize for workloads where most data on replicas is consistent.

Distributed cache consistency. Web-scale systems [11, 19, 34] frequently optimize for low latency and high availability at the expense of consistency. Many systems use caching to scale out to high throughput with low latency. While some systems use time-based mechanisms like leases [24] or TTLs [63] for cache consistency, others use invalidations [66] to keep caches up to date [11, 18, 56]. Even with invalidation streams, users are subject to replication lag without additional mechanisms for improved consistency. For example, FlightTracker [57] provides RYW consistency for users within a region by maintaining *tickets* representing the user's writes in a lookaside store. Similarly, Antipode [21] provides cross-datastore causal consistency by propagating *lineage metadata* alongside requests. These systems target different points in the consistency spectrum than Skybridge, but share common motivation in improving user experiences in a scalable manner.

Lu et al. [39] describe measuring consistency in Facebook's serving stack using both offline and online checking. Their offline framework reports a significantly lower rate of consistency violations vs. Skybridge's checker – this is likely due to their checker sampling user read requests (which may happen significantly later than writes) whereas Skybridge's checker injects read requests after 2 seconds. Their online checker injects reads similarly to Skybridge's checker and reports a significantly higher rate of inconsistency.

An and Cao describe *monotonic consistent caching* [2], a set of cache eviction policies that take consistency and atomicity into consideration. Like Skybridge, their lazy version selection strategy provides bounded staleness for users. MCC however is focused on cache evictions and targets lookaside cache systems whereas Skybridge targets systems with invalidation streams.

Bounded staleness variations. Bailis et al. describe *probabilistic bounded staleness* [7], a framework for users to determine "good enough" staleness for quorum-replicated datastores. Similarly, Ouyang et al. propose *probabilistic atomicity with well-bounded staleness* [48] for providing a probabilistic consistency level between eventual and strong consistency. While Skybridge does not guarantee bounded staleness by default due to rate limits, it seeks to provide as strong a bound as possible rather than forcing developers to reason about an acceptable probability of enforcement. Additionally none of these works target web-scale workloads like Meta. ACT [65] proposes a framework for application-specific consistency, but requires developers to augment applications to define the logical consistency units. There are other non-web workloads that benefit from bounded staleness, e.g., Cipar et al. [17] propose using bounded staleness for iterative AI algorithms.

8 Conclusion

In this work we introduced Skybridge, a system for providing bounded staleness for distributed in-memory caches. Skybridge acts as a staleness oracle, providing per-cache item staleness information when the DB replication stream is lagging. Skybridge is designed to complement the existing replication stream, optimizing for a flexible and compact replication format that enables real-time replication. Skybridge is efficient enough to enabled by default on billions of read requests per second, while simultaneously requiring only a small fraction of hosts. Skybridge is also effective, helping boost TAO's 2-second consistency up 99.99998%. Skybridge can provide significant developer and user experience benefits for Meta.

Acknowledgments

We would first like to thank the anonymous SOSP reviewers, OSDI reviewers and our shepherd Michael Stumm for their technical and editorial insights on improving the paper. We would also like to thank Audrey Chang, Aaron Kabcenell, Shilpa Lawande and Matt Pugh for their thorough review of the paper. Finally we would like to thank Amol Bhave, Arthur Charlton, Jinyu Han, Anwesh Joshi, Qiegang Long, Morgan McLain-Smith, Tushar Pankaj and Ilya Smirnov for their technical contributions to the systems in this paper.

References

[1] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal Memory: Defini-

tions, Implementation, and Programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] Shuai An and Yang Cao. Making Cache Monotonic and Consistent. *Proceedings of the VLDB Endowment*, 16(4):891–904, December 2022.

[3] Apache Software Foundation. Apache Kafka, 2024. https://kafka.apache.org/.

[4] Özalp Babaoğlu, Keith Marzullo, and Fred B Schneider. A Formalization of Priority Inversion. *Real-Time Systems*, 5(4):285–303, 1993.

[5] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 27–38, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Peter Bailis and Ali Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond: How Can Applications Be Built on Eventually Consistent Infrastructure Given No Guarantee of Safety? *Queue*, 11(3):20–32, March 2013.

[7] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, April 2012.

[8] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[9] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized Views Selection in a Multidimensional Database. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 156–165, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[10] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos Engineering. *IEEE Software*, 33(3):35–41, 2016.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, June 2013. USENIX Association.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2), June 2008.

[13] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proceedings of the VLDB Endowment*, 14(12):3014–3027, July 2021.

[14] Edith Cohen, Eran Halperin, and Haim Kaplan. Performance Aspects of Distributed Caches Using TTL-Based Consistency. *Theoretical Computer Science*, 331(1):73–96, 2005. Automata, Languages and Programming.

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, August 2008.

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems*, 31(3), August 2013.

[17] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 37–48, Philadelphia, PA, June 2014. USENIX Association.

[18] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, Boris Shkolnik, Sajid Topiwala, Alexander Pachev, Naveen Somasundaram, and Subbu Subramaniam. All Aboard the Databus! LinkedIn's Scalable Consistent Change Data Capture Platform. In *Proceedings of the Third*

ACM Symposium on Cloud Computing, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[20] Dormando. Caching beyond RAM: the case for NVMe, June 2018. https://memcached.org/blog/nvm-caching/.

[21] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 298–313, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[23] Google Cloud. Timestamp bounds | Spanner, 2024. https://cloud.google.com/spanner/docs/timestamp-bounds#bounded_staleness.

[24] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Operating Systems Review*, 23(5):202–210, November 1989.

[25] gRPC Authors. gRPC, 2024. https://grpc.io/.

[26] Jeffrey Helt, Abhinav Sharma, Daniel J. Abadi, Wyatt Lloyd, and Jose M. Faleiro. C5: Cloned Concurrency Control That Always Keeps Up. *Proceedings of the VLDB Endowment*, 16(1):1–14, September 2022.

[27] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[28] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-Based HTAP Database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, August 2020.

[29] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[30] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, April 1998.

[31] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In *Distributed Algorithms*, pages 126–140, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[32] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*, pages 17–32. Springer, 2014.

[33] Cockroach Labs. Cockroach DB - Load-Based Splitting, 2020. https://www.cockroachlabs.com/docs/stable/load-based-splitting.

[34] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[35] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[36] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[37] Federico Larumbe and Abhishek Mathur. Under the Hood: Broadcasting Live Video to Millions, December 2015. https://engineering.fb.com/2015/12/03/ios/under-the-hood-broadcasting-live-video-to-millions/.

[38] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, October 2012. USENIX Association.

[39] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 295–310, New York, NY, USA, 2015. Association for Computing Machinery.

[40] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[41] Microsoft, Inc. Consistency Level Choices - Azure Cosmos DB, 2024. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels#bounded-staleness-consistency.

[42] David L. Mills. RFC0958: Network Time Protocol (NTP), 1985.

[43] MySQL. MySQL, 2024. https://www.mysql.com/.

[44] MySQL. MySQL :: MySQL 8.0 Reference Manual :: 20.1.1.1 Source to Replica Replication, 2024. https://dev.mysql.com/doc/refman/8.0/en/group-replication-primary-secondary-replication.html.

[45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[46] Oleg Obleukhov and Ahmad Byagowi. How Precision Time Protocol is Being Deployed at Meta, November 2022. https://engineering.fb.com/2022/11/21/production-engineering/precision-time-protocol-at-meta/.

[47] Oracle. Range Optimization, April 2025. https://dev.mysql.com/doc/refman/8.4/en/range-optimization.html.

[48] Lingzhi Ouyang, Yu Huang, Hengfeng Wei, and Jian Lu. Achieving Probabilistic Atomicity With Well-Bounded Staleness and Low Read Latency in Distributed Datastores. *IEEE Transactions on Parallel and Distributed Systems*, 32(4):815–829, 2021.

[49] Lu Pan. Cache made consistent. In *Systems@Scale Summer 2022*, 2022. https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/.

[50] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. *Conflict-Free Replicated Data Types (CRDTs)*, pages 1–10. Springer International Publishing, Cham, 2018.

[51] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1135–1146, New York, NY, USA, 2013. Association for Computing Machinery.

[52] Redis, Ltd. Redis, 2024. https://redis.io/.

[53] Riak. Strong Consistency, 2024. https://docs.riak.com/riak/kv/latest/developing/app-guide/strong-consistency/index.html.

[54] Vinay Shankarkumar, Laurent Montini, Tim Frost, and Greg Dowd. RFC 8173: Precision Time Protocol Version 2 (PTPv2) Management Information Base, 2017.

[55] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[56] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 351–366, Oakland, CA, May 2015. USENIX Association.

[57] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 407–423. USENIX Association, November 2020.

[58] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and Practice of Bloom Filters for

Distributed Systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.

[59] Jeff Terrace and Michael J. Freedman. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, June 2009. USENIX Association.

[60] The Apache Software Foundation. Apache Zookeeper, 2020. https://zookeeper.apache.org/.

[61] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. *Proceedings of the VLDB Endowment*, 11(4):406–419, December 2017.

[62] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. IONIA: High-Performance Replication for Modern Disk-based KV Stores. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 225–241, Santa Clara, CA, February 2024. USENIX Association.

[63] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[64] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proceedings of the VLDB Endowment*, 16(12):3754–3767, August 2023.

[65] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000. USENIX Association.

[66] Haobo Yu, Lee Breslau, and Scott Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 163–174, New York, NY, USA, 1999. Association for Computing Machinery.

[67] Ping Zhang and Gary Chartrand. *Introduction to Graph Theory*, volume 2. Tata McGraw-Hill New York, 2006.

[68] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. *Proceedings of the VLDB Endowment*, 13(3):376–389, November 2019.



KPerfIR: Towards an Open and Compiler-centric Ecosystem for GPU Kernel Performance Tooling on Modern AI Workloads

Yue Guan[1†], Yuanwei Fang[2‡], Keren Zhou[3,4], Corbin Robeck[2‡],
Manman Ren[2‡], Zhongkai Yu[1†], Yufei Ding[1,2†], Adnan Aziz[2‡]
[1]*University of California, San Diego,* [2]*Meta,* [3]*George Mason University,* [4]*OpenAI*
[†]*{yueguan, zhy055, yufeiding}@ucsd.edu*
[‡]*{fywkevin, robeck, mren, adnanaziz}@meta.com*
kzhou6@gmu.edu

Abstract

In this work, we propose KPerfIR, a novel multilevel compiler-centric infrastructure to enable the development of customizable, extendable, and portable profiling tools tailored for modern artificial intelligence (AI) workloads on modern GPUs. Our approach integrates profiling capabilities directly into the compiler workflow, allowing profiling functionalities to be implemented as compiler passes, offering a programmable and reusable framework for performance analysis. This design bridges the gap between compilers and profilers, enabling fine-grained insights into complex optimization challenges such as overlapping the execution of fine-grained function units on GPUs. KPerfIR is integrated into the Triton infrastructure to highlight the power of a compiler-centric approach to advance performance analysis and optimization in the ever-evolving landscape of AI compilers. Our evaluation shows that our tool incurs low overhead (8.2%), provides accurate measurements (2% relative error), and delivers actionable insights into complicated GPU intra-kernel optimizations.

1 Introduction

In the era of artificial intelligence (AI) [27], the popularity of AI compilers has increased dramatically [7, 13]. Modern AI compilers, such as Triton [47], have become popular in bridging the gap between high-level machine learning framework operators (e.g. general matrix multiplications (GEMMs) [31], softmax [40], etc.) and the low-level, target-specific machine code [35]. The diversity of workloads and the rapid evolution of hardware architectures, particularly in GPUs, pose significant challenges in developing frameworks that are both modular in development and performant "out of the box" for users. Addressing these demands requires a flexible compiler infrastructure, like LLVM [25]'s Multi-Level Intermediate Representation (MLIR) [26] with a modular design, and Triton for its customizable high-performance AI operators for diverse workloads and backends.

Despite significant advancements, existing compilers often struggle to outperform hand-tuned implementations like cuBLAS [31], rocBLAS [6], and CUTLASS [45]. This limitation stems from GPU architectures' rapid evolution and the continuous emergence of novel operator variants. For instance, Nvidia's Hopper architecture incorporates advanced acceleration units such as 5th generation Tensor Cores (TC) [30] and Tensor Memory Accelerators (TMA) [15], which require sophisticated compiler passes to fully leverage their potential with a dataflow-oriented programming paradigm [15]. Meanwhile, the latest Flash-Attention-3 [41] (FA3) kernel, crucial for large language models (LLMs) [11, 46, 51], employs complex tiling and pipelining techniques that traditional compilers cannot effectively handle.

To enhance the compilers with novel compute paradigms and hardware features, profiling tools are critical in developing high-performance AI kernels and optimization passes. They are essential for identifying performance bottlenecks, analyzing execution flow, and understanding memory access patterns. Developers depend on these tools to pinpoint instruction stalls, optimize kernel performance, and refine compiler passes. Even for feedback-guided auto-optimization compilers [52], the design space consists of manually articulated transformation primitives and requires precise performance insights to guide effective optimization. Without effective profiling tools, achieving peak performance on modern GPU platforms becomes a daunting challenge.

However, existing profilers are poorly aligned with the advancements of compiler infrastructure to assist in the development of themselves and AI kernels. **Our key observation is that prior profiler designs are isolated from the compiler system and lack key connections to the upper-level framework operations.** Profiling tools are usually developed as external tools, detached from the intricacies of the compilation process, and fail to provide framework operation-informed, actionable insights tailored to the unique challenges of AI compilers. This disconnection severely hinders the ability of operator and AI compiler developers to enhance the performance of ML workloads.

To address this, we propose a novel compiler-centric infrastructure, KPerfIR, to facilitate the development of customiz-

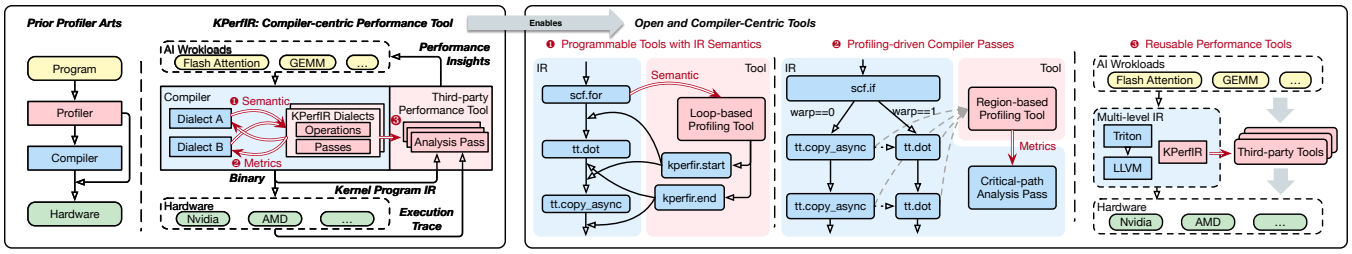


Figure 1: Concept of the KPerfIR infrastructure and ecosystem for compiler-centric performance tool. (Left) Overview and comparison of KPerfIR's compiler-centric design and prior profiler designs. (Right) Demonstrative examples of novel performance tools facilitated by the compiler-centric design of KPerfIR.

able and reusable performance tools[1], as illustrated in Fig. 1. Our approach establishes a seamless interplay between the compiler and profiler, enabling the exchange of program semantics and profiling metrics. Central to our methodology is incorporating multi-level compiler IR instrumentation into the profiling workflow. This allows performance tools to be composed as compiler passes with compiler-integrated profiling operations. This design fundamentally enhances performance tooling in three key ways.

❶ Programmable tools with IR semantics. Existing profilers have limited information on the program, restricting their ability to capture performance details tied to program semantics. For example, traditional tools cannot track how software pipelining [22] overlaps and evolves across pipeline stages due to missing loop-level information. By integrating IR-level instrumentation, KPerfIR enables profiling tools to capture these nuanced behaviors effectively.

❷ Profile-driven design of compiler passes. Many optimization techniques, such as autotuning [14, 43], rely on performance feedback to guide their tuning processes [52]. Traditional approaches treat compilers and profilers as isolated sub-modules, requiring standalone systems to coordinate optimization and evaluation. With KPerfIR, profiling passes can directly interact with compiler optimization passes, streamlining the feedback loop and enabling more optimizations.

❸ Reusable and extendable performance tools. By implementing tools native to the compiler IR, KPerfIR ensures their reusability and portability across frameworks and backends. Profiling tools can operate on shared upper-level representations used by AI frameworks and seamlessly lower to diverse backends, including Nvidia and AMD GPU platforms. Building on the MLIR philosophy of a shared set of infrastructure, we provide not only a powerful set of analysis tools but also a common set of building blocks and infrastructure for a community-driven ecosystem.

To showcase the capabilities of the KPerfIR infrastructure, we present the first region-based timing tool for GPUs to provide insights into intra-kernel behaviors. By leveraging the program's IR semantics, the tool efficiently updates the profiled regional timestamps to recover the inaccuracy caused by instrumentation, ensuring accurate profiling results while minimizing memory overhead. This level of precision was previously difficult to achieve because traditional profilers lack access to compiler-generated semantic information, such as loop structures and regional boundaries, making it challenging to attribute execution metrics accurately to high-level behaviors. This timing tool plays a crucial role in optimizing modern GPU kernels by enabling precise analysis of fine-grained overlapping and presenting an intuitive timeline visualization to guide performance optimizations.

We conducted an in-depth case study on the FA3 kernel. Leveraging the tool's fine-grained profiling results, we identified idle bubble regions in the baseline implementation and extracted key optimization insights. Based on these findings, we implemented novel compiler passes to enhance the kernel with an improved overlapping strategy, effectively reducing performance bottlenecks. In addition, we introduced performance modeling based on the profiling results to evaluate the efficiency of the optimized overlapping design.

This work makes the following key contributions:

- We propose KPerfIR, a compiler-centric infrastructure enabling customizable performance tools. Integrated into the Triton compiler, it supports AMD and Nvidia platforms, providing essential support for developers optimizing AI compilers and operators (Sec. 4).
- As a demonstration of the proposed infrastructure, we develop a novel region-based timing tool. This tool exemplifies the compiler-centric approach and offers fine-grained GPU profiling capabilities (Sec. 5).
- We perform an in-depth case study using the proposed timing tool to analyze GPU overlapping. The study reveals valuable insights into intra-kernel overlapping techniques, such as warp specialization (Sec. 6).

2 Background and Related Works

Although proposed methodologies support both Nvidia and AMD GPUs, we use the ones from Nvidia without any loss of generality unless discussing AMD platform-specific features. In the following, we describe the background of GPU compilers and profilers, focusing on their use in AI workloads.

[1]The source code is open sourced at https://github.com/triton-lang/triton/tree/main/third_party/proton/dialect.

Table 1: Comparison of GPU performance tools

Tools	Program IR	Intra-Kernel Profiling	Customized Regions	Platform Portability
NCU [33]	○	○	○	○
RocTracer [5]	○	○	○	○
AMD ATT	○	●	○	○
TorchProfiler [38]	○	○	○	●
Mosaic Profiler [10]	○	●	●	○
TK Profiler [44]	○	●	●	○
Tool with KPerfIR (§5)	●	●	●	●

2.1 GPU Compilers

The design of AI compilers has progressed significantly, with the introduction of MLIR [26] marking a pivotal step toward modularity, extensibility, and multi-layered abstraction. MLIR simplifies the complexity of modern computational workloads with a reusable framework that lowers the learning curve for compiler development while enabling custom components and third-party tool integration to drive innovation.

Building on the foundation of MLIR, Triton [47] is a groundbreaking compiler designed to bridge AI workloads and GPU hardware. With dialects specifically tailored for GPU programming, Triton offers high-level abstractions that simplify the development of efficient GPU kernels, addressing the needs of computationally intensive AI applications. Triton leverages MLIR's modular architecture to achieve reusability, seamless integration, and multi-level optimizations, exemplifying how domain-specific needs can be addressed within a unified framework. Triton's Intermediate Representations (IRs) encompass various dialects, including TTIR, TTGIR, Triton, TritonGPU, TritonNvidiaGPU, and TritonAMDGPU, each catering to specific stages of GPU programming.

2.2 GPU Profilers

GPU performance tools are indispensable for analyzing workload execution characteristics and identifying performance bottlenecks. They enable developers to understand application behavior and hardware utilization, which are critical for achieving optimal performance on GPU platforms. The GPU performance tools can be divided into two major categories.

Performance tools are ready-to-use profilers include NVIDIA's Nsight Compute (NCU) [33] and Nsight Systems (NSys) [34], as well as AMD's RocTracer and RocProfiler [4]. NCU provides detailed kernel-level analysis for CUDA applications, offering various metrics and actionable optimization suggestions. NSys delivers a holistic view of system-wide performance, capturing interactions between CPU and GPU to diagnose latency and synchronization issues. Similarly, RocTracer [5] supports performance optimization for AMD GPUs by tracing and analyzing applications in heterogeneous environments. An important aspect of these profiling tools is timing start and stop calls (e.g., cudaProfilerStart and cudaProfilerStop), which are implemented as host-side APIs *around* kernel launch calls and not directly within kernel code.

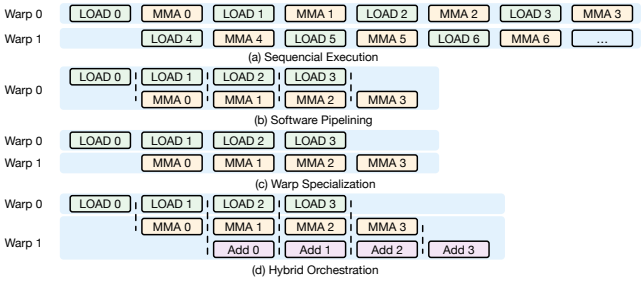


Figure 2: GPU overlapping techniques

MosaicGPU profiler [10], which inserts PTX instructions using high-level Python bindings, represents the closest idea to our approach. However, it operates at the assembly code level rather than on the high-level IRs, restricting its ability to provide comprehensive and reusable profiling capabilities. ThunderKitten (TK) [44], a promising DSL for the Nvidia platform, also developed a custom timing interface recently for region-based tracing. However, this tool is coupled with its DSL design and only supports the Nvidia platform.

Profiling infrastructures like CUPTI [32], NVBit [48], RocProfiler offer some level of instrumentation for customization. They allow developers to insert custom instructions at runtime, providing flexible profiling options. However, they are tightly coupled to the program and instrumentation framework [20]. This approach lacks portability and fails to integrate seamlessly with compiler workflows, limiting their ability to support cross-platform development.

2.3 GPU Overlapping Techniques

To effectively utilize GPU devices' resources and unleash their full potential, GPU kernels have to arrange diverse execution and memory units to overlap their execution to achieve better concurrency. With the introduction of dedicated on-chip acceleration units, such as Tensor Core [16] and Tensor Memory Accelerator [15], their overlapping is getting much more significant and difficult.

Wave Priority and Instruction Scheduling. The most straightforward solution for overlapping is to extend the scheduler with explicit priority setting. This is implemented as the wave priority technique mainly adopted by the AMD platform through compiler-based intrinsics (e.g., LLVM's `__builtin_amdgcn_s_setprio()`), which controls how wavefronts are scheduled and executed on the Compute Units (CUs) at compile-time. The CU will, at runtime, choose the wavefront with the higher priority when scheduling conflicting instruction types. By setting well-suited wave priorities, programmers can fully utilize the execution pipeline by explicit overlapping of memory loads on one wave and compute operations on another [3]. Overlapping can be further refined through compiler scheduling intrinsics such as `__builtin_amdgcn_sched_barrier()` to control

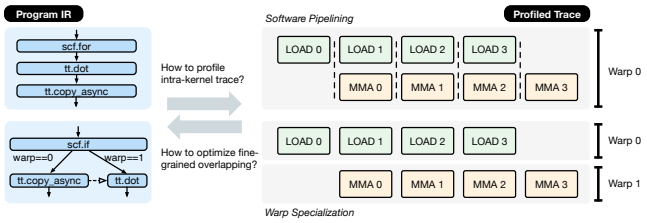


Figure 3: Motivating examples

the clustering and pipelining of matrix operations.

Software Pipelining (SWP) transforms the execution of independent loop iteration operations (i.e., memory and compute) into multiple stages to overlap between iterations, as shown in Fig. 2-(b). By using asynchronous instructions for data loading [8] and computation, different operations can be executed concurrently to overlap the latency. However, SWP requires extra scratchpad storage and registers in the loaded memory hierarchy to store the intermediate results for each stage. Software pipelining is usually adopted at multiple levels, such as global memory to shared memory and shared memory to register files, to hide the latency effectively. Besides the use for overlapping data loading and computation, SWP is also used for overlapping different sections of computations executed by separate hardware units. VALU, VMEM, and SMEM instructions [2, 3] can all be scheduled independently - the goal is to have these run entirely in parallel to tensor/matrix core operations. Due to its key advantage of enhancing utilization, SWP has gained widespread acceptance in high-performance operator libraries [45] and compiler optimizations [22].

Warp Specialization (WS) is another overlapping technique built with the producer-consumer model [8,23]. After splitting execution into stages, WS assigns stages to dedicated warps as producers and consumers. Instead of overlapping between different stages of loop iterations, WS overlaps between producer and consumer stages in different warps as depicted in Fig. 2-(c). WS was previously proposed as a solution for irregular and large problems to overlap unbalanced workloads and reduce register pressure [17]. Compared to SWP, where each warp acts as a producer and consumer, WS separates the roles so that registers can be assigned properly. For example, the data loading stage consumes much fewer registers compared to the heavy-lifting computation stage. However, this feature was not fully utilized due to the lack of hardware warp-level register allocation support. Recently, on the Nvidia Hopper architecture [15], novel features like register reallocation and asynchronous transaction barriers make WS achieve superior overlapping performance [42, 45].

3 Motivation

In this section, we analyze examples from current operator and compiler development that highlight the need for novel performance tooling, as illustrated in Fig. 3.

Fine-grained Comprehension. With the increasing complexity of heterogeneous execution units and asynchronous dataflow programming paradigms, it has become challenging to intuitively understand the execution pipeline. Users may choose SWP or WS to orchestrate data loading and computation for the cases shown in the example. However, to maximize hardware utilization, they must carefully determine stage partitions and the number of stages in SWP or decide execution orders and synchronization barriers in WS. Existing profilers provide aggregated results that lack the program semantics needed to correlate profiled metrics. To truly understand overlapping behavior, it is essential to parse the loop structure, analyze cross-warp dependencies, and track fine-grained metrics throughout the execution. Users struggle to identify and address inefficiencies effectively.

Takeaway 1: Performance profiling tools require the compiler's IR to provide fine-grained performance metrics.

Performance Optimization. Gaining a comprehensive understanding of execution behavior enables opportunities for manual optimizations. However, the challenge becomes even greater when implementing auto-optimizations within compiler passes. The compiler requires customizable performance feedback tied to the existing IR design to determine the optimal transformations. As shown in the example, selecting the appropriate overlapping design of WS and SWP requires stable performance estimations for each method. The compiler pass must rely on precise profiling data to guide its optimization choices effectively. A programmable performance tool embedded within the compiler infrastructure provides the essential first step in enabling this workflow.

Takeaway 2: Compiler optimization passes need programmable performance profiling tools to effectively guide their optimization decisions.

4 Compiler-centric Performance Tool

While it benefits compiler and kernel developers, the current compiler design lacks support for performance tooling. Existing MLIR dialects focus primarily on static, compile-time transformations and optimization passes, with little consideration for capturing dynamic traces or profiling semantics. This gap leaves developers without a native framework to analyze runtime behavior or fine-tune kernel performance within the compiler ecosystem.

Designing performance tooling infrastructure as MLIR dialects is a non-trivial task. For example, in Triton, such dialects must bridge the abstraction gap between the high-level tensor IR and the low-level target-specific IR while remaining clean abstraction to be consistent and compatible with existing multi-level program IR semantics. The ideal infrastructure should be able to handle diverse profiling demands with great extensibility for ever-evolving GPU architectures. Addressing this complexity, we introduce the KPerfIR dialect, the first of its kind, to integrate instrumentation passes directly

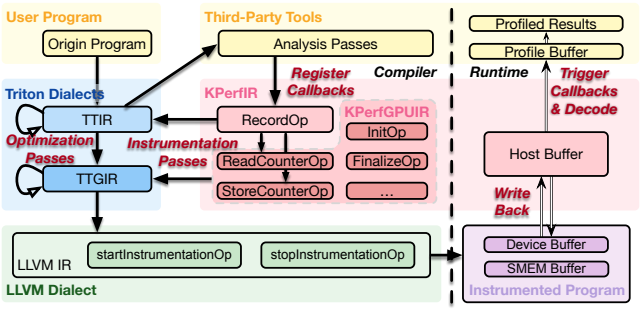


Figure 4: IR design and conversion passes

into the compiler IR as a performance tooling infrastructure. The system is implemented upon the mainstream AI compiler Triton [47], which adopts MLIR as its compiler infrastructure. We demonstrate the multi-level IR design and runtime convention of the KPerfIR infrastructure as shown in Fig. 4. We further discuss several representative use cases with KPerfIR.

4.1 IR Design

Within Triton's multiple-level IR structure, TTIR and TTGIR are the major program representations that developers interact with. We implement the KPerfIR as multi-layered MLIR dialects interacting with Triton's IR structure and link high-level MLIR operations to LLVM-IR level analysis. As shown in Fig. 4, we insert the record operations in MLIR dialects and lower them to architecture-specific TritonGPU operations. We also lower and instrument some scaffolding functions, such as data copy from local buffer to host memory, at the LLVM IR level. This architecture links the programming language constructs to hardware-related features for analysis. Specifically, we define novel operations at different levels to instrument profiling semantics and collect measurement data. The rewritten Triton program is then lowered into LLVM IRs for further analysis and code generation. This design balances programmability and complexity by higher-level IR design and lower-level code generation.

Instrumentation can be done at either the MLIR or LLVM IR level as a trade-off between flexibility and generality. MLIR level instrumentation can access high-level constructs, such as loops and data objects (e.g., tensors, matrices, etc.). However, the higher in the compiler pass pipeline instrumentation instructions are inserted, the more accuracy and reliability are influenced by the compiler backend's instruction scheduling and reordering. In contrast, LLVM IRs are closer to the low-level assembly code and provide more information about GPU devices. Nevertheless, instrumentation at the LLVM IR level may lose connection to high-level program semantics.

As such, we design the compiler-centric performance tool by linking the high-level MLIR dialects to LLVM IR level analysis as shown in Tbl. 2. We provide the profiling instrumentation interface at both the TTIR and TTGIR levels since they are general representations that hide low-level vendor-specific details and capture key optimizations such as software

Table 2: Main operations of KPerfIR

Operation	Inputs/Outputs	Attributes	Explanation
KPerfIR			
RecordOp	**In:** name, isStart	-	Main profiling IR.
KPerfGPUIR			
InitOp	**Out:** index_ptr	BufferType, BufferStrategy	Initialize and allocate memory.
FinalizeOp	**In:** index_ptr, global_ptr, data	-	Write back profiling to global memory and add metadata.
ReadCounterOp	**Out:** counter_ptr	MetricType, Granularity	Read a GPU metric counter into a scalar register.
StoreCounterOp	**In:** counter_ptr, index_ptr	isStart	Store a metric counter into a buffer.
LLVM			
startInstrumentationOp	**In:** instrumentation pass, index_ptr, buffer_size	-	Trigger low-level instrumentation.
stopInstrumentationOp	**In:** instrumentation pass	-	Stop low-level instrumentation.

pipelining, code motion, loop fusion, and warp specialization. KPerfIR is our highest abstraction level, which includes the key `RecordOp` operator, representing a general program marker, whose semantic interpretation solely depends on the KPerfIR to KPerfGPUIR lowering pass configurations. `RecordOp`'s inputs include `name` and `isStart`, specifying the annotation location and identifier in the program. An example of using `RecordOp` in the program is as follows.

```
kperfir.record <name, isStart=True>
// TARGET CODE REGION ...
kperfir.record <name, isStart=False>
```

Figure 5: An example of high-level record operations

The `RecordOp` at KPerfIR level abstracts away the hardware detail and is lowered to GPU-specific KPerfGPUIR operations. On the KPerfGPUIR level, we get vendor-independent but GPU-specific hardware features (e.g., shared memory abstraction). When lowering to KPerfGPUIR, various MLIR pass options are given to determine the conversion, such as the `MetricType` (specifying the performance counter) and the `Granularity` (e.g., warp-group, warp, thread, etc). For example, when profiling GPU cycles, each `RecordOp` is lowered to a `ReadCounterOp` and a `StoreCounterOp`, which can be scheduled separately by the compiler for the proper program location. `ReadCounterOp` collects the exact GPU cycle register values. `StoreCounterOp` stores the counter value into the profiling buffer with an offset. The corresponding buffer and bookkeeping resources are determined and allocated during the lowering pass given configurable constraints.

Tbl. 2 outlines key operators with inputs, outputs, and attributes presented in KPerfIR and KPerfGPUIR. The concrete operations instrumented are controlled by various MLIR pass options in the lowering/conversion passes, including `BufferType`, `BufferStrategy`, `MetricType`, `Granularity` and resource constraints (e.g., buffer size). Given these controlling knobs, the compiler generates specific KPerfGPUIR operations for performance tools whose profiling locations are specified by the `RecordOp` markers. For instance, specifying the `BufferStrategy` as circular, `MetricType` as clock, and `Granularity` as warp-group results in an intra-kernel profiler using a circular buffer to store the clock cycles of each warp-group (see

Sec. 5 for a detailed discussion of the profiler). In this case, the `StoreCounterOp` operation is converted to a `CircularStoreOp` to handle circular buffer event recording.

Besides, the compiler also generates scaffolding operations in KPerfGPUIR to setup and clean-up the environment of the profiling phase. `InitOp` initializes the profiling states (e.g., buffer index). We use stack allocation to enable the LLVM toolchain to apply register promotion for the buffer index, which sits in the critical path of clock measuring. This operation returns the address of the allocated index and the memory load/store operations will be optimized away by keeping the index value in the register with LLVM backend's optimization. `FinalizeOp` performs the clean-up and writes back the profiled records. Similarly, the exact buffer memory allocation operations (`LocalAllocOp`, `GlobalScratchAllocOp`, and `StackAllocOp`) are determined by the configurations during the lowering from KPerfIR to KPerfGPUIR as well[2].

Lastly, we have the LLVM level markers `startInstrumentationOp` and `stopInstrumentationOp` to control the low-level library-based instrumentation. The `startInstrumentationOp` tells the LLVM level to insert starting memory analysis/timestamp (including warp ID, SM ID). And the `stopInstrumentationOp` inserts ending memory analysis/timestamp and flushes data from the local buffer to the scratch buffer/host pinned memory. This allows instrumentation at lower IR levels to be linked to upper-level data objects and flow structures (e.g., inserting timestamps at the LLVM IR level around data objects and flow structures that only exist at higher IR levels).

Wrapping all instrumentation at the MLIR level makes the derived tools inextensible to ML frameworks due to their divergence on the supported operation set. In contrast, inserting instructions at the LLVM IR level allows for flexible and reusable tools. Analysis code can be written in any language, compiled to LLVM bytecode, and inserted into MLIR framework code. This feature is particularly attractive in MLIR frameworks that are not linked to Clang or its libraries but include C++ instrumentation functions. In other words, analysis functions can be written in HIP/CUDA and used in MLIR frameworks without C++ code (e.g., Triton, PyTorch).

4.2 Compiler Passes

We then leverage compiler passes to instrument the `RecordOp` to the target program and lower it to the underlying IRs. There are two main levels of compiler passes handling their insertion and conversion, including lowering KPerfIR into KPerfGPUIR and lowering KPerfGPUIR into LLVM.

The KPerfIR to KPerfGPUIR lowering pass converts the `RecordOp` to `ReadCounterOp` and `StoreCounterOp`, and inserts operations for resource allocation (e.g., `LocalAlloc`), and setup/clean-up operations (e.g., `InitOp` and `FinalizeOp`). At the entrance of the kernel function, we allocate the profiling buffer, global scratch memory, and bookkeeping resources. We defer the discussion of memory management in Sec. 5.2 and focus on the lowering pass explanation here. The allocated profiling buffer is associated with each `ReadCounterOp` to collect hardware performance counters (e.g., GPU cycle). The gathered raw data is written back to the GPU global memory by `FinalizeOp` in a predefined memory layout, which is inserted at the end of the kernel function. `FinalizeOp` has inputs including the number of the profiling records, base address of the profiling buffer, and base address of the profiler's global memory.

Triton lowers TTGIR into the LLVM backend to generate vendor-specific code (e.g., ptx and amdgcn) and perform low-level optimizations, such as redundant code elimination [18] and register promotion [28]. We develop a set of LLVM conversion patterns to handle the code generation of the instrumented profiling operations. Specifically, the `InitOp` is lowered to a stack allocation (`llvm.alloca`) of the buffer index. The `ReadCounterOp` is lowered to a read of a performance counter (e.g., %clock) to the register, and the `StoreCounterOp` is lowered to a register value store with tag creation and buffer index management. For the `FinalizeOp`, we compute the global memory offset for the current thread block and assign the first thread as a worker to write the entire profiling buffer back to the global memory.

We provide principal support regarding backends to reduce the interference of the high-level instrumentation in optimizations such as instruction re-ordering. For Nvidia GPUs, the impact is negligible since the hardware is responsible for instruction scheduling with the PTX mostly following the instrumented program with KPerfIR's operators associated. Instruction will not be scheduled around critical instruction like WGMMA. AMD GPUs expose instruction scheduling to software, even for instruction SMEM load and MFMA in amdgcm, making the instrumented operator a key factor. As such, we provide three levels of configurations adjustable by users: 1. manual adjusting with KPerfIR hints; 2. direct instrumentation on amdgcn, and 3. specifying instruction scheduling window with barrier mask explicitly.

4.3 Working with Third-Party Tools

We then demonstrate the interfaces for users to develop their third-party tools shown on the right side in Fig. 4, including the instrumentation APIs handling rewriting and the necessary memory management to decode the profiled raw data.

Instrumentation APIs. To build a customized performance tool with the KPerfIR dialect, the user must explicitly modify the kernel signature and update the IR with instrumentation passes. We provide two APIs for the user to interact with the KPerfIR, the command-line API and the Python API. With the command-line API, all Triton functions encountered at runtime with the command-line API are instrumented with the

[2]Please refer to the documentation for more details at https://triton-lang.org/main/dialects/ProtonOps.html.

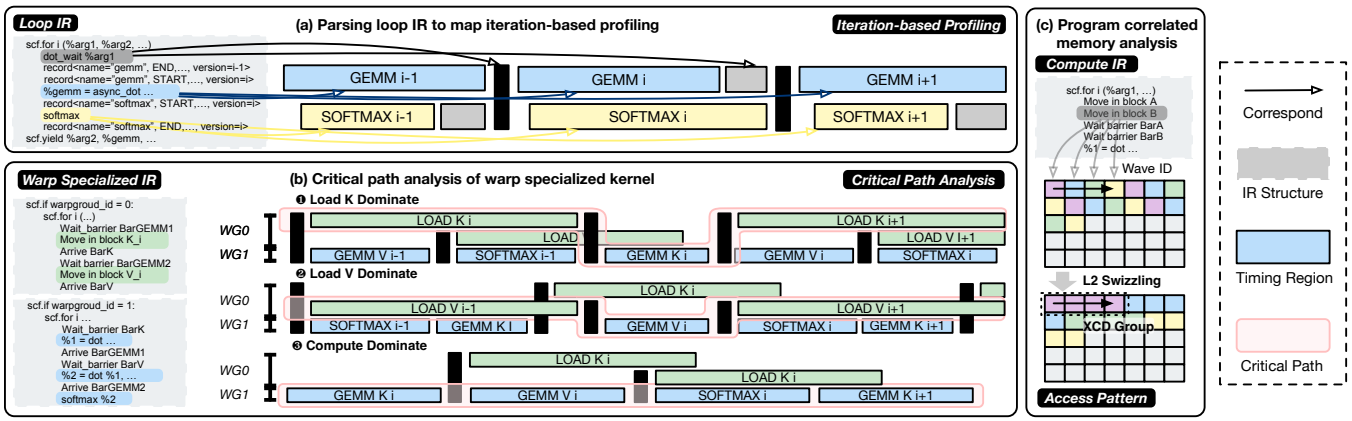


Figure 6: Novel use cases facilitated by KPerfIR's compiler-centric approach

specified analysis passes. This allows for handy manipulation of the target workload but lacks flexibility since users cannot skip kernels they are not interested in.

The Python API lets the user specify the instrumentation entry and the target function to be instrumented. The instrumentation entry here refers to the insertion position in the compilation process (i.e., before/after which pass). On the other hand, the instrumentation point specified by the `RecordOp` refers to the position in the IR. It encompasses the core API as `KPerfIR.patch(instrumentation_obj, fn)`, where `instrumentation_obj` identifies the compiler pass to be instrumented (either before or after) and `fn` is optional to select the instrumented kernel. If the `instrumentation_obj` specifies MLIR instrumentation, we adopt MLIR's pass infrastructure `PassManagement`. We will instrument LLVM at the end of the compilation passes. Additionally, we provide a `KPerfIR.unpatch()` API to restore all existing instrumentation. This involves maintaining the kernel's original and instrumented version within the KPerfIR runtime.

Runtime Memory Management. Another important part of the third-party tool involves the management of buffer storage and profiling results. At the top level, we rewrite the kernel signature with an extra last argument as a pointer to the device buffer and modify the calling convention. The accumulated profiling data is then returned to the host buffer managed by the KPerfIR runtime. Here, we adopt a discarding-based circular buffer design, which will be elaborated with an actual tool design in Sec. 5.2. We have two ways to manage GPU device buffer and host buffer communication, following prior studies [53, 54] The first approach is to use CUDA-managed allocation so that the device buffer is allocated in a memory space visible to both the GPU and the CPU. The second approach is to allocate a priority stream with a higher priority than the one used by PyTorch's runtime and use that stream to copy the data back to the CPU.

Once the data is copied back to the CPU, it is decoded into a similar format to the CUPTI Activity API as a C/C++ struct. Third-party tools will register a callback to process this data when the runtime is initialized. Callbacks are triggered once the host buffer is decoded to allow these tools to process the data, like CUPTI activities. As such, the tools can post-process the data to compatible formats with their own front-end visualizers, such as Chrome Trace [19] or Hatchet [9].

4.4 Use Cases

We then showcase several novel performance tools facilitated by KPerfIR infrastructure as shown in Fig. 6.

Iteration-based Timing. Understanding the synchronous behavior of warp groups across iteration boundaries is crucial for analyzing SWP. For instance, observing overlapping at specific iteration steps requires associating the loop induction variable `i` with a version argument in the timing records. However, traditional profiling tools lack access to the loop semantics necessary to determine iteration positions, making it challenging to comprehend the precise overlapping behavior of resources across iterations. By incorporating profiling passes into the compiler, we can instrument the IR to collect clock ticks alongside loop iteration index semantics, enabling post-processing to construct fine-grained timelines, as shown in Fig. 6-(a). While tools like MLIR's Python bindings can implement this functionality, they are ad-hoc and restricted to simple loop structures, underscoring the need for deeper integration of compiler semantics into profiling workflows.

Critical-path Analysis. Runtime feedback is invaluable for compiler passes, particularly for identifying execution critical paths in scenarios involving WS. For example, in the FA3 kernel, producer warp groups load input tensors, while consumer warp groups execute multiple GEMM operations and a softmax computation. The critical path—determined by factors like data movement and computation latency—varies with program tiling and access patterns, making static analysis insufficient for arrangement. For the FA3 case, there are three common critical paths dominated by different operations, as shown in Fig. 6-(b). By embedding profiling passes into the compiler, we can instrument timing records to capture runtime

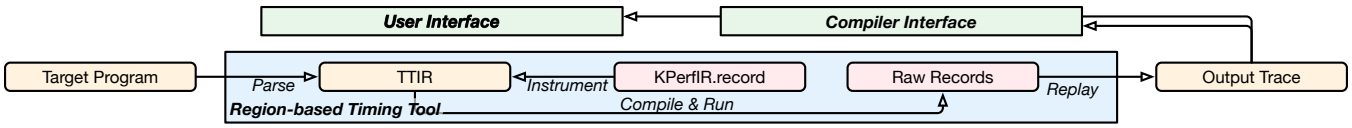


Figure 7: Workflow of the region-based timing tool

stage latencies and dynamically identify the critical path. This enables the compiler to strategically insert asynchronous barriers, reducing waiting bubble times and improving resource utilization. Such optimization passes highlight the value of integrating performance tooling with compiler passes.

Program Correlated Memory Analysis. Fine-grained memory profiling benefits significantly from a compiler-centric design, allowing developers to correlate memory performance metrics, such as access patterns, heat maps, and bank conflicts, with high-level program objects. Taking Fig. 6-(c) as an example, analyzing the memory access pattern of a specific data object, such as a tensor, requires parsing the program IR to insert profiling intrinsics that map high-level constructs to low-level hardware registers. This approach is particularly critical for optimizing L2-level scheduling, such as with AMD's chiplet designs [1]. Profiling warp-level memory access patterns and correlating them with program objects enables the design of optimized swizzling strategies for L2 access. Such capabilities, achievable only through compiler-centric profiling, demonstrate the necessity of integrating memory profiling tools into the compiler infrastructure.

5 The Region-based Timing Tool

In this section, we dive into a powerful third-party performance tool built upon the KPerfIR infrastructure that conducts region-based intra-kernel timing to assist in comprehending and optimizing the overlapping GPU hardware resources. To build a customized tool with KPerfIR, the users must implement the instrumentation passes, the profile data management, and the post-processing process as introduced in Sec. 4.3. These are elaborated with the region-based timing tools as its workflow for instrumentation (Sec. 5.1), its memory system (Sec. 5.2), and a trace replay post-processing (Sec. 5.3).

5.1 Workflow

We first introduce the workflow of the timing tool with two kinds of interfaces, the user interface and the compiler interface, as shown in Fig. 7. Depending on how the target program is parsed and the profiling regions are specified, different interfaces are invoked to instrument the profiling records. KPerfIR will then compile and execute the instrumented program to collect the raw performance counters. At last, we replay the trace with the raw metadata and produce the final timeline trace. This trace is output and feedback to different interfaces for visualization or profiler-guided compiler passes.

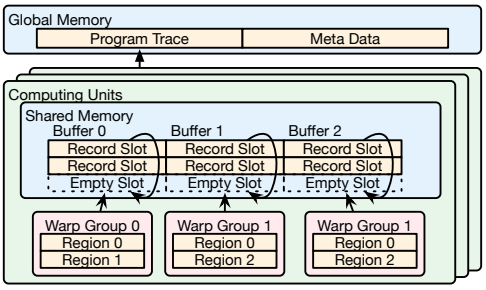


Figure 8: Memory management of the region-based tool

User interface is the basic usage for developers to profile interested regions of a target GPU program. We provide PythonDSL bindings in Triton kernels and also allow manual rewriting and overriding the dumped middle-level IR for the concerned regions. It is straightforward as the developer manually rewrites the dumped middle-level IR for the concerned regions. This is extremely useful when developing high-performance kernels and debugging efficiency problems. The KPerfIR will instrument the original program with rewritten program IR and produce the output trace. Developers can utilize visualization tools, such as Chrome Tracer [19], to get an intuitive view of the program timeline.

Compiler interface is how the compiler interacts with the profiler. Compared to the user interface, the compiler passes are responsible for parsing the middle-level IR and insert profiling record at concerned regions. Similarly, the instrumented program is compiled and executed, and the profiled traces are feedback to the compiler passes within the compiler.

5.2 Profile Data Management

We then introduce the specific management and decoding techniques of the tool. The system accommodating the storage at each memory hierarchy is shown in Fig. 8. We demonstrate the data structure and layout bottom-up in the following.

Warp-level Region Data Structure. For each fine-grained profiling region, there are two profiling records inserted, the region start record and region end record. Each record has 8 bytes, containing a 4-byte tag and a 4-byte payload, as shown in Fig. 9. The tag contains 1 flag bit as *START* or *END*, and 31-bit control bits that are dependent on the backend will be explained later. The payload contains a 32-bit time stamp metric captured from the hardware counter (cycle for %clock in Nvidia and the LSB 32 bits of `S_MEMTIME` instruction in AMD). This design achieves a good trade-off between space capacity and write speed. Because the storage consumption is

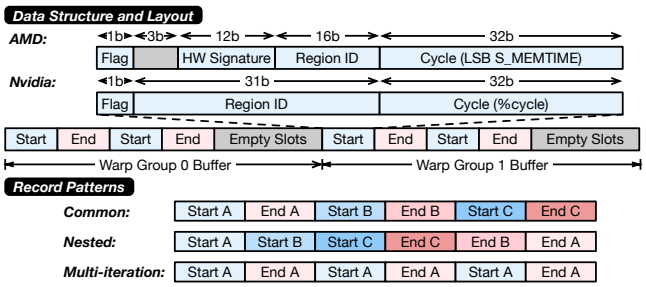


Figure 9: Circular buffer and record buffer

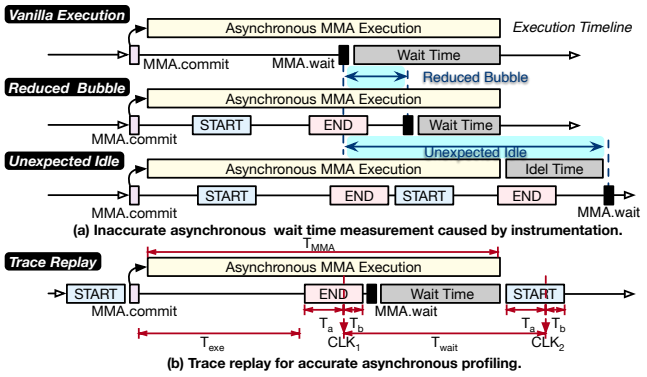


Figure 10: Trace replay

low and the store takes a vectorized store instruction.

We use a 32-bit clock to capture the cycle tick of the current record, which may cause value overflow. We address this in the post-processing procedure, where we detect and throw exceptions on overflows. The trace replay can compensate for the clock wrap-around overflow as long as each iteration runs less than 4 billion cycles (4 seconds under 1GHz). Since most of the execution is spent on loops, the restriction is relaxed to the loop iteration level, which is less than 1 millisecond for most cases, making 32-bit cycle data a safe choice. If necessary, KPerfIR could easily extend to a 64-bit clock by adding dedicated operators and attributes in the IR.

SM-level Data Layout. The record is then copied to the data buffer in shared memory from the local register file with a store instruction. The data buffer is split into profiling spaces by warp groups with non-overlapping record slots, as shown in Fig. 8. As such, the record index at compile time determines the store location offset in shared memory. For example, if we allocate 64 slots (0.5KB) and each thread block contains 2 warp groups (8 warps, 256 threads), then each warp group has 32 slots. During compile time, we pre-compute the base address of each profiling space and generate code to manage the indexes for each warp group.

The memory layout of each warp group buffer is shown in Fig. 9. Start and end records are interleaved in shared memory to support nested regions and multiple iterations. The lower part of the figure illustrates three profiling patterns: common, nested, and multi-iteration. Records are stored without pairing and aligned during trace replay to ensure correctness based on the profiling patterns.

Shared Memory Circular Buffer. Because shared memory has limited capacity, it may not be able to accommodate all profile data, especially with a long multi-iteration record pattern. For example, with 2 warp groups, 4 profiled regions, and a loop with 512 iterations, it takes up to 4096 record slots, translating to a 16KB shared memory storage. For the production-level kernels, we usually get the available shared memory capacity left ranging from 1KB to 4KB for intra-kernel profiling's data buffer, only up to 1.75% of a total of 228KB on H100 device [15].

We propose a *circular buffer* structure for each warp group's data buffer as a solution balancing accuracy and usability. The circular buffer keeps only the trace's tail record cyclically when reaching the shared memory limit. We overwrite the oldest results cyclically instead of flushing them to the lower-level memory. The insight is that visualizing a few recent iterations is sufficient to identify the bottleneck. We wrap around the index with a pre-calculated buffer capacity to implement the circular buffer. If the captured events are overflowing the buffer, it is redirected to an occupied slot. This causes extra instructions for index management, which is addressed during compile-time. We only need lightweight modular instructions to round the index at runtime.

Besides, this *circular buffer* is the default memory management behavior in the runtime. The user can switch to the naive flush strategy, where the captured events are instantly written back to the global memory when the buffer is fully occupied. This strategy can keep all the profile events with the overhead of many more frequent memory write instructions. In the KPerfIR's infrastructure, we provide a comprehensive operation to support the memory allocation at each hierarchy, including `Stack`, `Shared`, `Global`, and management strategies, including `Circular` and `Flush`. Users can make use of these abstractions to develop their tool, combining the use of the buffers and management strategies.

Specifically, we implement a collaborative store strategy for the AMD backend to minimize profiling overhead. Since only one timestamp record is profiled per warp or warp group, AMD GPUs utilize branching with thread masks to issue instructions with predication. However, this thread divergence can lead to unexpected instruction cache misses, causing overheads up to 600 cycles. To mitigate this, we enforce all threads to write to the same shared memory location, retaining only the last result. To align the records, we include an additional 12-bit signature derived from the least significant bits (LSB) of the `HARDWARE_ID REG`, as shown in Fig. 9. This signature encodes the `wave_slot_id`, `SIMD_id`, and `pipe_id` to annotate the profiled thread. At the warp group level, we ensure all four warps write to the same shared memory location.

Global Program Trace. Lastly, we gather the profiling data from all SMs back to the GPU global memory before returning from the kernel function. We keep the temporary profile

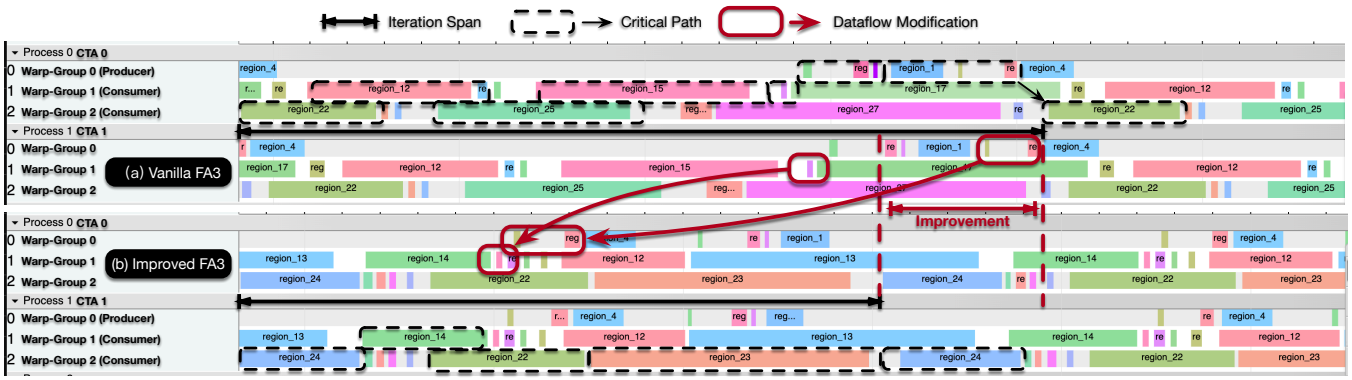


Figure 11: Region-based timing results for FA3 kernels and overlapping improvements guided by profiling

data in the shared memory and copy the entire data buffer at once by the end of the kernel. Several extra metadata are attached for reconstructing the trace as shown in Fig. 8, including thread block index, warp-level indices, and a number of recorded slots in each warp group. The Triton frontend automatically patches the kernel function with an extra argument (`profile_mem`), allocating the GPU scratch memory for such profile data.

5.3 Trace Replay

The major challenge in the region-based timing scenario of intra-kernel profiling is the perturbations caused by the profiling instructions. Post-processing techniques, such as periodic sampling [21], are usually used to mitigate the inaccuracies by sampling the performance metrics periodically and summarizing the results afterward to reduce the overhead associated with data collection. In the region-based timing tool, we propose a similar post-processing method *trace replay* to rescale the profiled regions according to the program semantics owing to the KPerfIR's design. We can get accurate visualized results using this trace replay mechanism by offsetting a constant to account for the clock measuring overhead.

For regions with only synchronous instructions, we can easily rebuild the region by subtracting the overhead of the record operation. For asynchronous instructions, we have two kinds of common inaccuracies as shown in Fig. 10. In the vanilla execution timeline example, we have a sequence of successive instructions executed after issuing an asynchronous MMA operation on the Tensor Core. After the program reaches the barrier in the instruction stream, there is a period of waiting time before the functional unit can produce results. To understand the asynchronous behavior here, users can insert profiling records in the successive execution to better align the operations. However, the introduction of such records causes inaccurate wait time measurements. In the *reduced bubble* case, the wait time is underestimated due to the extra overhead caused by profiling. Things are even worse in the *unexpected idle* case, where the functional unit's execution time cannot cover the profiling overhead. An unexpected idle time will be observed, and the optimization will be misleading.

Table 3: Profiled regions of the FA3 kernel

Tag	Function	Region ID	
		Vanilla	Improved
Load K	Load the K tensor.	3	3
Load V	Load the V tensor.	6	6
GEMM0	Compute QK on TC.	12,22	12,22
GEMM1	Compute PV on TC.	14,24	15,25
Softmax	Compute softmax on CUDA core.	13,23	17,27

To address this, we propose deducting the accurate wait time after profiling in the trace replay procedure as shown in Fig. 10-(b). Instead of placing one `END` record after the asynchronous instructions, we insert two `START` records before the asynchronous launch and after the wait barrier and one `END` record right before the barrier. This guarantees an accurate wait time T_{wait} measurement as $T_{wait} = (CLK_2 - T_a) - (CLK_1 - T_a) = CLK_2 - CLK_1$, where T_a and T_b are the elapsed time in the record region before and after the clock read instruction respectively. The profiling overhead is canceled out in the post-processing with the two carefully placed records. We can derive the T_{exe} time similarly. This method requires the $T_{MMA} - T_{exe} > T_a + T_b$. Since the profiling overhead is less than 25 cycles for most cases, as shown in Sec. 6.4, the execution time of the functional unit can cover only the record inserted here, which is usually about 1000 cycles.

6 Evaluation

In this section, we evaluate the performance of the designed region-based timing tool and the KPerfIR approach.

6.1 Experimental Setup

Testbed. The evaluation of KPerfIR was conducted on servers equipped with state-of-the-art GPU cards, including NVIDIA H100-HBM3 and AMD MI300X. The software environment included Triton 3.0.0 and LLVM 19.1, ensuring compatibility with the latest compiler technologies and GPU architectures.

Benchmarks. To assess the usability and performance of the proposed tool, we benchmarked it on key AI workloads, specifically GEMM [36] and experimental Flash Attention

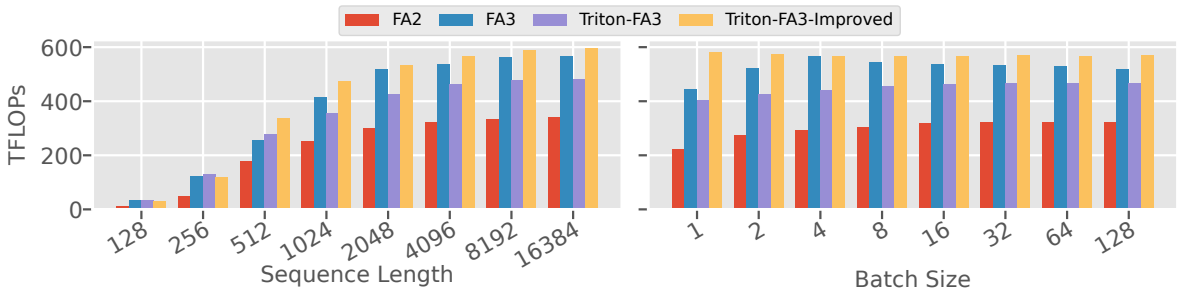


Figure 12: Benchmarking FA3 kernels with a head dimension of 128 and 16 heads. The batch size and sequence length are set to 16 and 4096 in the two benchmarks, respectively.

operators from Triton [29]. These operators are foundational to popular AI models, representing the majority of workloads in tasks such as training and inference. We selected mainstream implementations of these operators that deliver state-of-the-art performance and conducted sensitivity experiments to evaluate the method's effectiveness and scalability.

6.2 A Journey of Flash-Attention 3

We first demonstrate utilizing the region-based timing tool to improve the FA3 kernel. As FA3 involves many optimization techniques dedicated to the Nvidia platform, we focus on optimizing the H100 GPU first and conduct many more thorough benchmarks later.

6.2.1 Attention Overlapping Optimization

We begin by demonstrating the usage of the region-based timing tool to identify a particular inefficiency in the FA3 kernel's WS overlapping design. Using the proposed timing tool, we parse the FA3 kernel's IR structure and profile its asynchronous behavior to guide a better overlapping design. The critical stages of the FA3 kernel are identified and listed in Tbl. 3. Each stage is labeled with a unique region ID, providing a clear mapping for analysis. By employing the timing tool, we obtain a fine-grained timeline trace of the vanilla FA3 kernel from Triton, shown in Fig. 11-(a), which also illustrates the user interface with the Chrome Trace as the front-end. The trace identifies the critical path consisting of 4 GEMMs and 2 loading stages, including region 22, 12, 25, 15, 6 and 3. Specifically, the loading V stage of region 6 is blocked by the arrival barrier of region 16 in consumer 1, causing a longer critical path and lower hardware utilization.

Following this observation, we can advance the arrival barrier of V in region 16, as shown with the red arrows, to prevent it from stalling the issue of successive data loading. The GEMM1 regions are released from the critical path by fully overlapping it with the concurrent K tensor loading. This modification breaks data dependency between the arrival and computation and requires extra pre-loading in the prologue before the iterations. We can achieve a much more compact timeline where the softmax and GEMM computation are overlapped, shown in Fig. 11-(b). The improved overlapping results in a reduced wall time of each iteration in the kernel.

Table 4: Performance models

Category	Analytic Model
SWP Model	$\Delta = N_{WG} * N_{pipe} * \Sigma_i T_{comp} - Max_i(T^i_{load} + T^i_{comp})$ $\begin{cases} \Sigma_i T^i_{comp} * N_{loop}, \Delta >= 0 \\ Max_i(T^i_{load} + T^i_{comp}) \times N_{loop} / N_{pipe} \end{cases}$
WS Model	$\Sigma_{i \in CriticalPath} T^i_{load/comp}$
Compute Model	$FLOPs/Throughput$
Memory Model	$T_{read} + Bytes/Bandwidth$

6.2.2 Performance Modeling for GPU Overlapping

We then demonstrate how to adopt the proposed compiler-centric profiler design to build a compiler pass to determine the optimal overlapping design. This is achieved by extracting the critical path of the FA3 kernel with the profiling results and compute the utilization of the workload using an overlapping performance model. The performance model is outlined in Tbl. 4, focusing on compute and data loading stages while simplifying less relevant aspects such as initialization, epilogue, and overall kernel performance, which have been studied extensively in prior works [22].We emphasize the non-trivial parts of modeling the overlapping techniques.

For the SWP model, the discriminant Δ determines whether the bottleneck lies in data loading or computation. If $\Delta \geq 0$, the data loading latency is fully overlapped by computation, and the total latency corresponds to the accumulated computation time across all stages. Conversely, if $\Delta < 0$, the latency is dominated by the most time-consuming loading and computation stages. For the WS model, the latency is determined by identifying the kernel's critical path, as discussed in Sec. 4.4. The WS latency is intuitively calculated as the sum of the latencies of all stages along the critical path.

Using this model, we can quantitatively assess the overlapping efficiency of the FA3 kernel. The region-based profiler collects detailed performance data, allowing us to analyze

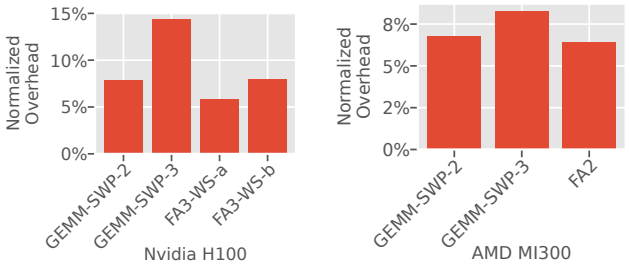


Figure 13: Normalized latency overhead

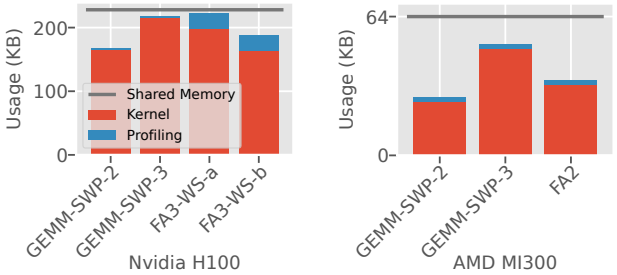


Figure 14: Memory usage

the current implementation and refine overlapping designs to minimize latency and improve resource utilization. For example, for an attention kernel with head dimension 128, head number 16, batch size 16, and sequence length 4096, the 2-stage SWP is calculated with a 467.07 TFLOPs and the vanilla Triton FA3 reaches a 526.97 TFLOPs. With the optimization passes incorporated, we can tune an FA3 kernel with improved overlapping, achieving 582.44 predicted TFLOPs.

6.2.3 Improved FA3 Evaluation

Combining the manual optimization passes with the profiling insights and performance model pass together, we can get an improved FA3 compiler pass optimization suite built upon Triton. Compared to the original execution dataflow of the vanilla Triton FA3, the optimized FA3 kernel achieves a 24.1% improvement as shown in Fig. 12. This shows the efficacy of the optimization insights discovered with the assistance of the performance tool. The improved Triton-FA3 kernel also outperforms the best manual FA3 kernel [41] by 7.6%. Particularly, without the assistance of the region-based timing tool, the vanilla Triton-FA3 kernel fails to achieve a competitive performance compared to the manual kernels. This suggests the immeasurable value of a fine-grained profiling tool benefiting the compiler passes and kernel optimization.

6.3 Profiling Overhead

We evaluate the profiling overhead of the proposed region-based timing tool in terms of both latency and memory consumption. Particular attention is given to shared memory usage, which is a critical bottleneck in the design of performance tools for GPU workloads. We benchmark SOTA GEMM operators with SWP designs divided into 2 or 3 stages, marked as GEMM-SWP-2/3, and WS FA3 operators with vanilla overlapping and the improved counterpart with the profiling insights, marked as FA3-WS-a/b.

Latency Overhead. We measure the end-to-end latency of the instrumented kernels and normalize the results to their original execution time. Fig. 13 illustrates the latency overhead across the evaluated benchmarks. For most cases, the overhead remains under 10%, ensuring that the proposed tool is practical for real-world scenarios without significantly impacting kernel performance. For the most complicated SWP GEMM kernel with three stages, we insert many records to cover its three stages. Even in this case, the overhead is kept within 15%. This minimal latency impact is crucial for enabling real-time profiling and maintaining production-level throughput during optimization workflows.

We hightlight that with the post-processing trace replay technique in Sec. 5.3 mitigating the interference of the profiling instructions, we can get relatively accurate profile results even if the overhead is extended. We have also introduced several memory management techniques like circular buffers, which together reduce the profiling overhead to approximately 8% in our evaluation. Moreover, ongoing enhancements, such as low-level instruction scheduling and clock offsetting, can further reduced the overhead.

Shared Memory Overhead. As detailed in Sec. 5.2, the proposed timing tool employs a circular buffer design to store recent profiling records within the limited shared memory available on GPUs. This approach ensures compatibility with diverse workloads, regardless of the size of the available shared memory, while also allowing flexible adaptation of the profiling life cycle.

Fig. 14 shows the shared memory usage across the benchmarked workloads. For this experiment, the circular buffer size was set to either maximize usable shared memory space or accommodate all profiling records for the workload. The results demonstrate that the tool effectively operates within the constraints of shared memory, even for industrial-grade kernels, without spilling profiling records to external memory.

For instance, in the most storage-intensive kernel, the SWP GEMM with 3 stages, there remains an unused shared memory space of 10.9 KB. With 4 profiled regions, the timing tool can track up to 16 iterations, providing sufficient and stable coverage to estimate the entire execution process. This efficient memory usage highlights the applicability of the tool for large-scale GPU workloads.

6.4 Low-level Deep-dive

To provide an empirical analysis of the optimization performance degradation introduced by KPerfIR, we examine its im-

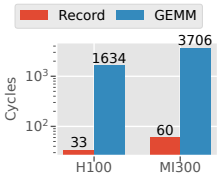


Figure 15: Cycle-level benchmarks

	GEMM	Theoretical	Actual
Active Cycles	199381	224981	229663
Relative Performance	0.89	1	1.02

Table 5: Performance degradation evaluation

pact at the low-level instruction level. Specifically, we identify the instrumented instructions generated by IR-level profiling and compare them with their vanilla counterparts to evaluate how profiling IR affects the compilation stack.

Cycle-level Overhead. At the GPU assembly code level, the SASS ISA, each KPerfIR record node is lowered to three instructions: a clock read instruction, an integer move instruction, and a predicated store instruction. These instructions collectively incur an average overhead of 33 cycles, as shown in Fig. 15. This latency represents the per-record profiling overhead observed in our sensitivity analysis. For loop-based timing, where profiling records are inserted into loop structures, five additional instructions are generated at the loop entry for index management. The instructions for each profiling record remain consistent at three, with their destination adjusted dynamically based on the loop iteration variable.

Optimization Degradation. One critical observation in our analysis is that profiling instrumentation can interfere with the compiler's low-level optimizations, such as instruction reordering and constant folding. Integrated profiling semantics at the IR level inherently trade some low-level control for improved tool portability and compatibility, as discussed in Sec. 4. This tradeoff introduces potential risks of unintended instruction reordering when profiling instructions interact with adjacent non-profiling instructions.

To study this impact, we modeled the theoretical execution time of the instrumented kernel as

$$T_{theoretical} = T_{vanilla} + N_{record} * Cycle_{record}. \quad (1)$$

The N_{record} is the number of instrumented records, and the cycles are measured as Fig. 15. Using this model, we compared the theoretical execution time with the actual performance of the instrumented workload to assess deviations caused by profiling instrumentation.

The results, summarized in Tbl. 5, show that the performance impact remains within 2%. This suggests that the profiling instrumentation at the compiler IR level introduces minimal overhead while maintaining compatibility with compiler optimizations. The findings highlight the practicality of the proposed approach, ensuring accurate profiling without significant performance degradation.

7 Discussions

7.1 Limitations

One limitation of our approach stems from restricted visibility into certain vendor-specific performance counters. Since KPerfIR inserts profiling instructions directly into the program to collect runtime information, the set of accessible metrics is constrained by the vendor's ISA. In contrast, vendor-provided tools such as NVIDIA's Nsight Compute (NCU) and AMD's ROCm Tools (e.g., ATT) have privileged access to undocumented performance registers not exposed to third-party developers. These proprietary counters can only be utilized through their official APIs or libraries. While our infrastructure may not achieve full parity with these tools in terms of raw counter access, our IR-level instrumentation enables unique insights into program behavior otherwise difficult to obtain with closed-source profilers.

7.2 Workload Generality

Beyond AI Workloads. While KPerfIR currently targets AI workloads through its integration with the Triton compiler, the core ideas behind our approach, namely a multi-level IR and transformation passes for performance tools, are broadly applicable. These techniques can generalize to other domains and compilers, such as those used in high-performance computing (HPC) [39] or scientific simulation [24].

Distributed Workloads. KPerfIR is also amenable to distributed GPU workloads [49, 50], which are typically structured as a series of kernels involving both computation and communication. For workloads that launch computation and communication kernels separately, KPerfIR already provides native support. For fused kernels, where computation and communication are fused within a single kernel [12, 37], KPerfIR can theoretically instrument and analyze them as well. However, full integration depends on ongoing upstream efforts to extend Triton's support for such fused distributed execution.

8 Conclusion

As AI compilers and GPU architectures continue to evolve, performance tooling must also advance to meet the growing demands of modern compilers. To address this need, we introduced KPerfIR, a novel compiler-centric profiling infrastructure built upon the Triton compiler. KPerfIR bridges the gap between compiler and profiler design, enabling the development of programmable, reusable tools and unlocking new possibilities for performance analysis and compiler optimizations. We envision KPerfIR as a foundation for an open, compiler-centric ecosystem, empowering the community to construct diverse and innovative performance tools that adapt to the dynamic needs of AI workloads.

Acknowledgement

The authors would like to thank the anonymous reviewers for their constructive feedback on improving the work. We also thank our shepherd for the support during the revision process. We would like to thank our colleagues from Meta: Bert Maher, Hongtao Yu, Taylor Robie, Elliot Gorokhovsky and OAI people Philippe Tillet, Thomas Raoux, Pawel Szczerbuk, for their early discussion and feedback on this project. This work was inspired by Adam Paszke's work on MosaicGPU, presented both at Meta and Triton Conference. This work was supported in part by NSF 2124039. Keren Zhou's work is supported by a donation from AIGCSEMI LLC.

References

[1] Advanced Micro Devices, Inc. *AMD CDNA 3 Architecture*, 2024.

[2] Advanced Micro Devices, Inc. *"AMD Instinct MI300" Instruction Set Architecture*, 2024.

[3] Advanced Micro Devices, Inc. Composable kernel (CK) library, 2024.

[4] Advanced Micro Devices, Inc. *ROCm ROCProfiler*, 2024.

[5] Advanced Micro Devices, Inc. *ROCm ROCTracer*, 2024. Version 6.2.4.

[6] AMD. *rocBLAS Library*, 2023.

[7] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[8] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Seattle Washington, November 2011. ACM.

[9] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. Hatchet: Pruning the overgrowth in parallel profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–21, 2019.

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[11] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. Flux: fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[14] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[15] Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 43(3):9–17, 2023.

[16] Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and programmability. *Ieee Micro*, 38(2):42–52, 2018.

[17] Neal C. Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W. Keckler. WASP: Exploiting GPU Pipeline Parallelism with Hardware-Accelerated Automatic Warp Specialization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–16, Edinburgh, United Kingdom, March 2024. IEEE.

[18] Jack W. Davidson and Christopher W. Fraser. Eliminating redundant object code. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 128–132, New York, NY, USA, 1982. Association for Computing Machinery.

[19] Google. Chrome trace format, 2023.

[20] Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, et al. Amanda: Unified instrumentation

framework for deep neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1–18, 2024.

[21] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. Profile inference revisited. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.

[22] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs, May 2023. arXiv:2210.16691.

[23] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 680–694. Curan, 2023.

[24] David E Keyes, Lois C McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, et al. Multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications*, 27(1):4–83, 2013.

[25] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[28] John Lu and Keith D. Cooper. Register promotion in c programs. *SIGPLAN Not.*, 32(5):308–319, May 1997.

[29] Meta. *Experimental FlashAttention3 using Triton*, 2024. Version 2024.12.2.

[30] NVIDIA Corporation. *NVIDIA Turing GPU Architecture Whitepaper*, 2018.

[31] NVIDIA Corporation. *cuBLAS Library*, 2023. Retrieved from https://docs.nvidia.com/cuda/cublas/.

[32] NVIDIA Corporation. *CUPTI: CUDA Profiling Tools Interface*, 2023.

[33] NVIDIA Corporation. *NVIDIA Nsight Compute*, 2024. Version 2022.4.

[34] NVIDIA Corporation. *NVIDIA Nsight Systems*, 2024. Version 2024.7.1.

[35] NVIDIA Corporation. *NVIDIA PTX*, 2024. Version 8.5.

[36] OpenAI Corpora. *Group GEMM in Triton*, 2024. Version 2024.11.

[37] Kishore Punniyamurthy, Khaled Hamidouche, and Bradford M Beckmann. Optimizing distributed ml communication with fused computation-collective operations. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2024.

[38] PyTorch Core Team. PyTorch Profiler. https://pytorch.org/tutorials/intermediate/profiler_tutorial.html, 2021. Accessed: 2025-04-21.

[39] Daniel Reed, Dennis Gannon, and Jack Dongarra. Reinventing high performance computing: challenges and opportunities. *arXiv preprint arXiv:2203.02544*, 2022.

[40] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[41] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

[42] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[43] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[44] Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

[45] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[46] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.

[47] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[48] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.

[49] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, et al. Wlb-llm: Workload-balanced 4d parallelism for large language model training. *arXiv preprint arXiv:2503.17924*, 2025.

[50] Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. Rap: Resource-aware automated gpu sharing for multi-gpu recommendation model training and input preprocessing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 964–979, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564*, 2023.

[52] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[53] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. Gvprof: A value profiler for gpu-based clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2020.

[54] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. Valueexpert: Exploring value patterns in gpu-accelerated applications. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–185, 2022.

A Artifact Appendix

Abstract

The KPerfIR is a performance tool infrastructure for the Triton [47] compiler and the results for the OSDI'25 submission are derived from some feature branches. While the core concept is stable, the implementation is still evolving and subject to change. A formal documentation can be found at https://triton-lang.org/main/dialects/ProtonOps.html.

Scope

We showcase the usability of the performance tool and the improvement results discussed in the paper by reproducing Fig. 11 and Fig. 12. Specifically, profiling the FA3 kernel with the region-based performance tool demonstrate the usage of the compiler-centric design. A comparison of the profile trace from the vanilla and improved FA3 kernels shows the methodology of intra-kernel region profiling. And the evaluation results shows the actual performance improvement.

Contents

We provide a Docker image that contains the Triton compiler and evaluation scripts for the artifact evaluation. Please follow the installation instructions in the following section to set up the environment. Within the Docker image, the contents are organized as follows:

```
workspace/
—— triton            % Triton compiler source code
—— tritonbench       % Triton FA3 benchmark suite
—— kperfir_artifact  % Other scripts and files
```

Hosting

This artifact is open sourced at https://github.com/ChandlerGuan/kperfir_artifact with the main branch and commit version e45891d.

Requirements

The Docker environment requires a Linux-based system with NVIDIA Container Toolkit installed. While the KPerfIR project is designed to be cross-platform, the artifact evaluation for the FA3 kernel requires a Nvidia H100 GPU.



Mirage: A Multi-Level Superoptimizer for Tensor Programs

Mengdi Wu Xinhao Cheng Shengyu Liu[†] Chunan Shi[†] Jianan Ji
Man Kit Ao Praveen Velliengiri[‡] Xupeng Miao[♯] Oded Padon[◇] Zhihao Jia

Carnegie Mellon University *Peking University*[†]
Pennsylvania State University[‡] *Purdue University*[♯] *Weizmann Institute of Science*[◇]

Abstract

We introduce Mirage, the first multi-level superoptimizer for tensor programs. A key idea in Mirage is μGraphs, a uniform representation of tensor programs at the kernel, thread block, and thread levels of the GPU compute hierarchy. μGraphs enable Mirage to discover novel optimizations that combine algebraic transformations, schedule transformations, and generation of new custom kernels. To navigate the large search space, Mirage introduces a pruning technique based on abstraction that significantly reduces the search space and provides a certain optimality guarantee. To ensure that the optimized μGraph is equivalent to the input program, Mirage introduces a probabilistic equivalence verification procedure with strong theoretical guarantees. Our evaluation shows that Mirage significantly outperforms existing approaches even for DNNs that are widely used and heavily optimized. Mirage is publicly available at https://github.com/mirage-project/mirage.

1 Introduction

Enabling high-performance execution of deep neural networks (DNNs) on GPUs is critical for modern ML applications. Today's DNN frameworks generally specify DNN computation using tensor programs, which are directed acyclic graphs whose nodes and edges represent tensor algebra operators (e.g., matrix multiplication) and tensors (i.e., n-dimensional arrays) shared between operators.

To optimize an input tensor program, existing frameworks (e.g., PyTorch [34] and TensorFlow [9]) use manually designed rules to map the tensor program to expert-written GPU kernels. These approaches generally require extensive engineering efforts to design and implement optimization rules, and they may miss certain optimization opportunities. To address these challenges, recent work has introduced *automated* approaches that optimize tensor programs by searching over a comprehensive space of program transformations and applying them based on their performance on target GPUs. These approaches generally fall into two categories.

The first category of work, including Halide [35], TVM [13], and Ansor [51], is motivated by the idea of algorithm and schedule separation[1] introduced in Halide and optimizes the *schedule* of a tensor program while fixing the algorithm. For a given algorithm, these optimizers automatically generate performant kernels by searching for possible strategies to execute the kernel on the target hardware. However, due to the linear algebra nature of DNNs, a tensor program can be represented by a wide spectrum of mathematically equivalent algorithms. Existing schedule-based optimizers only consider kernels whose algorithms are manually specified by users, resulting in missed optimization opportunities.

The second category of work, including TASO, Grappler, Tensat, and PET, considers *algebraic transformations*, which exploit mathematical equivalence among different algorithms for a tensor program [3, 25, 46, 48]. Examples of algebraic transformations include (1) converting one linear algebra operator into another, such as transforming a convolution to a matrix multiplication; (2) fusing multiple operators to reduce memory access and kernel overhead; and (3) reorganizing operators based on commutativity, associativity, and distributivity. These optimizers perform algebraic transformations at the algorithm level and require programmers to manually specify the set of available operators and their implementations. They are thus limited by the performance of the provided kernels.

All existing automated optimization approaches, from both categories, still require programmers to manually specify a set of kernels (each defined by a tensor function), and then explore the search space of algebraic *or* schedule transformations. However, some advanced performance optimizations require coordinated transformations across the kernel, thread block, and thread levels of the GPU compute hierarchy, and involve introducing completely new kernel computations (e.g., a custom kernel that decomposes standard kernels and fuses only certain computations). Such optimizations are not included in the search space of existing automated methods and must still be implemented manually.

[1] In the schedule optimization literature, an algorithm describes what to compute in a kernel and a schedule specifies how to compute the kernel.

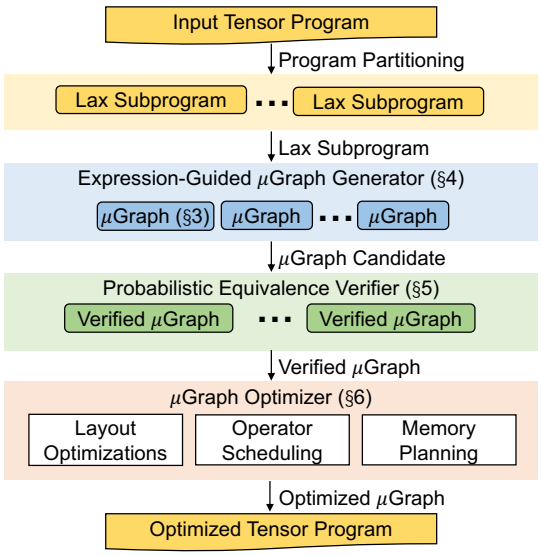


Figure 1: An overview of Mirage.

One such example is FlashAttention [17] (see §8.2 for details), which optimizes attention [47] on GPUs by reordering operators at the algorithm level (algebraic transformations), reorganizing the computation across GPU kernels (yielding new custom kernels), and adapting the parallelization strategy of each kernel to the GPU architecture (schedule transformations). The transformations required for this example cannot be automatically discovered by existing frameworks and must therefore be implemented manually. An implementation of FlashAttention in Triton [43], a widely used tensor program optimizer, contains more than 700 lines of code [8].

We present Mirage, the first *multi-level superoptimizer* for tensor programs. Mirage automatically discovers and verifies sophisticated optimizations of tensor programs that require joint optimization of algebraic transformations, schedule transformations, and the discovery of new custom kernels.

A key idea in Mirage is μGraphs, a *hierarchical graph representation* that specifies tensor programs across multiple levels of the GPU compute hierarchy. By uniformly treating the kernel, thread block, and thread levels, μGraphs can capture both algebraic and schedule transformations across these levels. Moreover, optimizing a μGraph can introduce new custom kernels, which go beyond both algebraic and schedule transformations. For example, Mirage automatically discovers the μGraphs representing FlashAttention [17] and its inference variant FlashDecoding [5] as well as other μGraphs that outperform these manually designed kernels by up to 2.2× for certain use cases. Most of these optimizations discovered by Mirage are outside the search space of existing methods.

Figure 1 shows an overview of Mirage. Mirage first splits an input tensor program into subprograms that fall into the restricted LAX fragment. The LAX fragment, formally defined in §5, includes multi-linear operators such as matrix multiplication and convolution, division (useful for normalizations), and limited exponentiation (useful for activations). Partitioning a tensor program into LAX subprograms reduces the optimization search space while preserving most optimization opportunities; it also enables Mirage's probabilistic equivalence verifier.

Expression-guided μGraph generator. For each LAX subprogram, Mirage's *expression-guided generator* exhaustively searches for possible μGraphs equivalent to it. A key challenge Mirage must address is its significantly larger search space compared to prior superoptimization techniques. For example, TASO [25] and PET [46] search only for tensor programs at the kernel level, using a fixed set of pre-defined kernels, while Mirage considers superoptimization across the kernel, thread block, and thread levels. To efficiently navigate this significantly larger search space, Mirage introduces a novel pruning technique based on *abstract expressions*, which greatly reduces the number of μGraphs Mirage must consider while providing a certain theoretical guarantee on the optimality of the discovered μGraphs. Mirage further reduces the search space by focusing the search on the kernel and block levels and using a rule-based approach for the thread level.

Probabilistic equivalence verifier. For a μGraph discovered by Mirage, verifying its functional equivalence with the input program introduces another challenge, since the input and output tensors of a program include up to many millions of elements. A key idea behind Mirage is *probabilistic equivalence verification*, which performs random tests over finite fields to check equivalence between μGraphs. While random tests typically provide limited correctness guarantees for general programs, Mirage leverages a novel theoretical result showing that the restrictions imposed by the LAX fragment ensure that, for LAX programs, random tests over finite fields offer strong correctness guarantees. Specifically, we show that a polynomial identity testing (PIT) algorithm [37, 54] can be generalized to LAX programs, yielding a randomized algorithm for LAX program equivalence that can be made arbitrarily precise. Mirage uses this randomized algorithm to (probabilistically) ensure that each optimized program is equivalent to the input program.

μGraph optimizer. For each verified μGraph, Mirage's *μGraph optimizer* maximizes its runtime performance by further considering potential tensor layouts, scheduling operator execution orders, and planning memory allocation at all of the kernel, thread block, and thread levels. Finally, Mirage returns an optimized tensor program based on the best discovered μGraph for each individual LAX subprogram.

Evaluation results. We evaluate Mirage on a variety of commonly used DNN benchmarks on NVIDIA A100 and H100 GPUs. Even for DNN benchmarks that are widely used

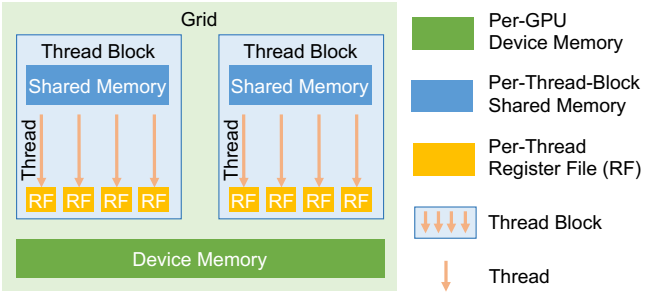


Figure 2: GPU compute and memory hierarchy.

and heavily optimized by existing systems, such as the group-query attention used in LLMs [41], Mirage still outperforms current approaches by up to 3.3× by exploiting subtle custom kernels and optimizations missing in existing systems.

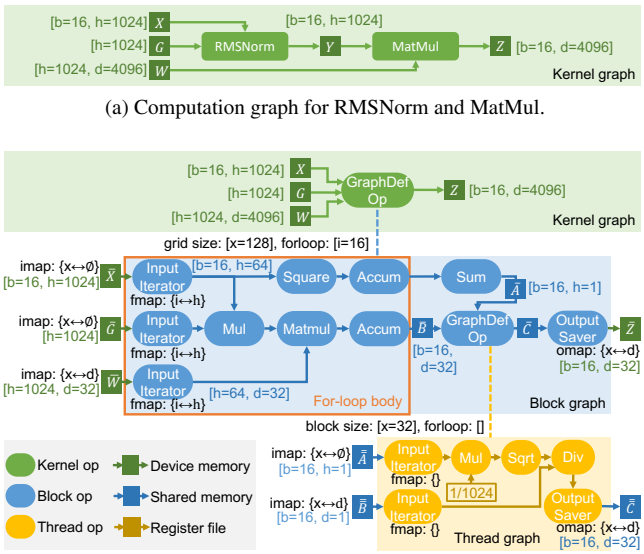
2 Multi-Level Graph Representation

Mirage uses a μGraph to specify the execution of a tensor program on GPUs. A μGraph contains hierarchical graphs at multiple levels to represent computation at the kernel, block, and thread levels[2]. This section first describes the GPU hierarchy and uses Figure 3 as a running example to introduce the key components of a μGraph.

GPU hierarchy. Figure 2 shows the hierarchy of today's GPUs. Computations on GPUs are organized as *kernels*, each of which is a function executed simultaneously on multiple GPU cores in a single-program-multiple-data (SPMD) fashion. A kernel includes a grid of *thread blocks*, each of which is executed on one GPU streaming multiprocessor and includes multiple *threads* to perform computation on individual data elements. Each thread is associated with a per-thread *register file*, and all threads within a thread block can access *shared memory* to enable collective operations. Finally, all inputs and outputs of a kernel are stored in GPU *device memory*.

Kernel graph. Each tensor program corresponds to one *kernel graph*, where each node represents a kernel running on an entire GPU, and each edge is a tensor shared between kernels. All tensors in a kernel graph are stored in GPU device memory since different kernels cannot share data in register files or shared memory. Each node in a kernel graph can be a *pre-defined* kernel operator supported by existing kernel libraries such as convolution by cuDNN [15] and matrix multiplication by cuBLAS [16]. In addition, to enable fine-grained inter-kernel optimizations such as kernel fusion, a node in a kernel graph can also be a *graph-defined* kernel operator, whose semantic and behavior are defined by a lower-level (i.e., block) graph. As an example, the kernel operator in Figure 3b is a graph-defined operator specified by a block graph.

[2]For simplicity, we use the term *block* to refer to a thread block of a CUDA kernel and *thread* to refer to a single CUDA thread.



(a) Computation graph for RMSNorm and MatMul.

(b) The best μGraph discovered by Mirage.

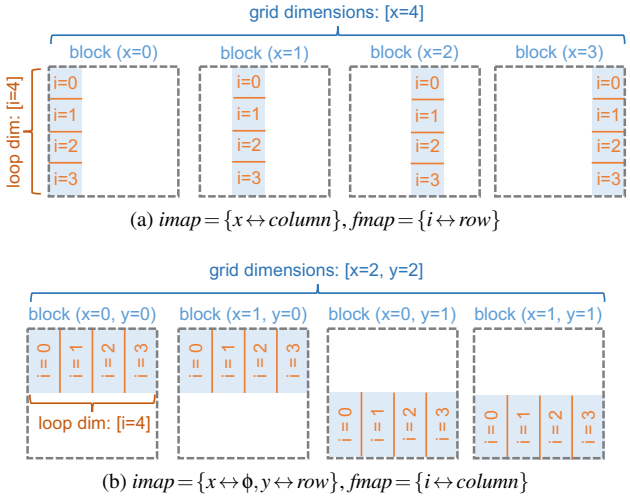
Figure 3: Figure 3a is the computation graph for RMSNorm and MatMul. Figure 3b shows the best μGraph discovered by Mirage for computing RMSNorm and MatMul, which fuses the computation in a single kernel to reduce device memory access and kernel launch overhead, outperforms existing approaches by 1.9×. Numbers in brackets indicate tensor shapes, and numbers in braces show the *imap*, *omap*, or *fmap* for the corresponding operators.

Block graph. A *block* graph specifies computation associated with a thread block[3], where each node denotes a *block operator* specifying computation within a block, and each edge (blue arrows in Figure 3b) is a tensor shared between block operators. Mirage stores all intermediate tensors within a block graph in GPU *shared memory* for two considerations. First, GPU shared memory offers much higher bandwidth than device memory, and this design allows Mirage to reduce device memory access by maximally saving intermediate results in shared memory. Second, for tensors whose sizes exceed shared memory capacity and must be stored in device memory, Mirage uses these tensors to split computation into multiple block graphs, each of which only contains tensors in shared memory. This separation does not introduce additional access to device memory.

Each block graph is also associated with properties specifying its execution, which we introduce below.

Grid dimensions. All blocks within a kernel are organized into a mesh with up to 3 dimensions, identified as x, y, and z. A

[3]In the CUDA programming model, a kernel's computation is defined as computations for independent thread blocks.



(a) $imap = \{x \leftrightarrow column\}, fmap = \{i \leftrightarrow row\}$

(b) $imap = \{x \leftrightarrow \phi, y \leftrightarrow row\}, fmap = \{i \leftrightarrow column\}$

Figure 4: Demonstrating how an input tensor is partitioned across blocks and for-loop iterations with *imap* and *fmap*.

block graph is associated with up to three *grid dimensions* that specify the number of blocks along the x, y, and z dimensions. The block graph in Figure 3b launches 128 blocks.

First, for each input tensor to a graph-defined kernel operator (e.g., X, G, and W in the kernel graph in Figure 3b), the associated block graph contains an *imap*, which specifies how the input tensor is partitioned into sub-tensors for individual blocks. For each grid dimension (i.e., x, y, or z), the *imap* maps it to either (1) a data dimension of the input tensor or (2) a special *replica* dimension ϕ. For (1), the mapped data dimension is *equally partitioned* across blocks along the grid dimension. For (2), the input tensor is *replicated* across these blocks. For example, the block graph in Figure 3b takes three inputs—$\overline{X}$, $\overline{G}$, and $\overline{W}$—representing the input tensors to each block. For $\overline{W}$, its $imap = \{x \leftrightarrow d\}$ indicates that the d dimension of tensor W is partitioned into 128 equally sized chunks. As a result, $\overline{W}$ has shape $[h = 1024, d = 32]$.

Second, for each output tensor of a block graph (e.g., $\overline{Z}$ in Figure 3b), the block graph includes an *omap*, which specifies how the outputs of all blocks are concatenated to construct the final output of the kernel operator. In an *omap*, each grid dimension must map to a data dimension of the output tensor, since different blocks must store disjoint tensors in device memory. For $\overline{Z}$ with shape $[b = 16, d = 32]$ in Figure 3b, its $omap = \{x \leftrightarrow d\}$ indicates that blocks with the same x index are concatenated along the d dimension, resulting in a tensor Z with shape $[b = 16, d = 4096]$.

For-loop body. To fit large input tensors in shared memory and to overlap data loading from device memory with computation, a block graph can include a *for-loop body*, which is executed multiple times to complete a kernel. Often, the for loop in a kernel is followed by some post-processing. For example, when computing an average value, the for loop would perform the summation of n values and the post-processing would divide by n. Mirage specifies the for-loop body of a block graph using *input iterators*, *for-loop accumulators*, and all operators in between, as shown in the orange box in Figure 3b). Each input tensor to a block graph first passes through an *input iterator*, which loads part of the tensor (e.g., $\overline{X}$, $\overline{G}$, and $\overline{W}$) from device memory into shared memory. Each input iterator is associated with an *fmap* to specify which part of the input tensor to load in each iteration. Formally, the *fmap* maps each for-loop dimension to either (1) a data dimension of the input tensor or (2) the replica dimension ϕ. Similar to *imap*, the tensor is equally partitioned along that dimension for (1) and replicated for (2). Figure 4 shows how an input matrix is partitioned across blocks and for-loop iterations using different *imap* and *fmap*.

Each block graph is also associated with a *for-loop dimension*, which determines how many iterations the for-loop body is executed to complete the kernel. In addition, Mirage uses *for-loop accumulators* (e.g., the two `Accum` operators in Figure 3b) to accumulate intermediate results computed in each iteration (using standard accumulators, e.g., summation and max) and store the accumulated results in shared memory. Once the for-loop body is completed, Mirage proceeds to execute the remaining operators outside the for-loop body directly on the accumulated results. An *output saver* then saves the final result from shared memory back to device memory.

Thread graph. A *thread graph* further reduces computation scope from a block to a single thread. Similar to a block graph, each thread graph is also associated with *block dimensions*, which specify the organization of threads within the block, and *for-loop dimensions*, which define the total number of iterations to finish the defined computation. Each thread graph includes *input iterators*, each of which loads an input tensor (e.g., $\overline{\overline{A}}$ and $\overline{\overline{B}}$ in Figure 3b) from shared memory into register files, and *output savers*, each of which stores an output tensor from register files back to shared memory (e.g., $\overline{\overline{C}}$). A thread graph is the lowest-level graph in a μGraph and contains only pre-defined thread operators.

Tensor layout. Each tensor in the kernel, block, or thread graph is associated with a *tensor layout* (omitted in Figure 3 for simplicity), specifying how the tensor is linearized in memory. Note that tensor layouts affect only the performance of a μGraph and have no impact on its output correctness.

Definition 2.1 (μGraph Validity)**.** A μGraph G is *valid* if: (1) for each kernel, block, and thread operator $o \in G$, its input and output tensors match the specification of o; (2) all tensors in each kernel, block, and thread graph can reside in GPU device memory, shared memory, and register file, respectively; and (3) for each block and thread graph with a for-loop body, any path from an input to an output passes through exactly one input-iterator, one for-loop accumulator, and one output saver.

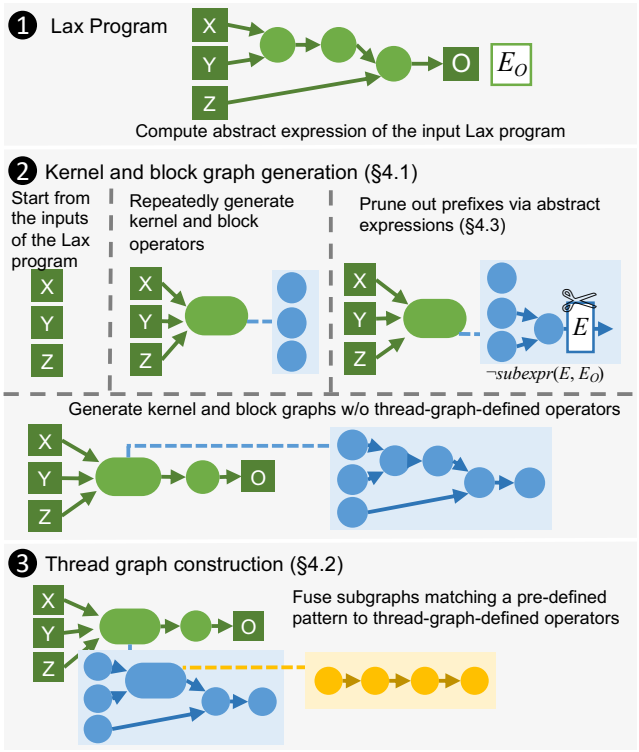


Figure 5: An overview of the μGraph generator.

Comparison with prior work. Prior work separately considers algebraic [25, 46] or schedule transformations [13, 31, 35], while μGraphs can represent both in a uniform way. Specifically, the grid and for-loop dimensions and their corresponding mappings (i.e., *imap*, *omap*, and *fmap*) to tensor dimensions constitute a comprehensive search space of possible schedules for graph-defined operators. The hierarchical graphs across the kernel, block, and thread levels allow Mirage to explore algebraic transformations at these levels.

3 Case Study: RMSNorm

In this section, we use root mean square layer normalization (RMSNorm) [50] as a case study to demonstrate the advantages of the μGraph representation and Mirage's superoptimization approach. RMSNorm is a widely used normalization technique in recent large language models [41]. Formally, RMSNorm takes two tensors, X and G, as inputs and normalizes their element-wise products according to the root mean square:

$$Y_{ij} = \frac{X_{ij}G_j}{\text{RMS}(X_i)}, \text{RMS}(X_i) = \sqrt{\frac{1}{d}\sum_{j=1}^{d} X_{ij}^2}, \quad (1)$$

where d is the hidden dimension size of X.

RMSNorm is often followed by a matrix multiplication (MatMul). Figure 3a shows the computation graph of an RMSNorm followed by a MatMul operator, where X is the input tensor, and G and W denote two weight tensors. Existing ML compilers generally launch two separate kernels for RMSNorm and MatMul computations, since both operations internally perform reductions across an input dimension, making it challenging to fuse their computations into a single kernel. This approach requires storing intermediate results (i.e., Y) in device memory since different kernels cannot share data in shared memory or register files.

Figure 3b shows the best μGraph automatically discovered by Mirage for computing RMSNorm and MatMul in a single kernel. The computation is fused in a single graph-defined kernel operator to avoid saving intermediate results (i.e., Y) in device memory and reduce kernel launch overheads.

We highlight the key differences between the μGraph discovered by Mirage and the original μGraph. These differences involve discovering new custom kernels and combining algebraic and schedule transformations, making it infeasible to discover the final μGraph by separately considering algebraic and schedule transformations. First, Mirage reorders MatMul and the division of RMSNorm by leveraging the commutativity of matrix multiplication and element-wise division (algebraic transformation). Second, Mirage performs the accumulation in the root mean square (i.e., $A_i = \sum_j X_{ij}^2$) and the accumulation in the matrix multiplication (i.e., $B_{ik} = \sum_j X_{ij}G_jW_{jk}$) in parallel (schedule transformation), avoiding writing the accumulation results to device memory. Next, Mirage instantiates a thread graph to perform a sequence of element-wise operators while maintaining all intermediate results in register files (schedule transformation). Finally, the best discovered μGraph uses a new custom kernel to fuse the computation of RMSNorm and MatMul, reducing device memory access and kernel launch overheads. This μGraph outperforms the hand-written kernels in existing systems by 1.5× and 1.9× on NVIDIA A100 and H100 GPUs respectively.

4 Expression-Guided μGraph Generator

This section introduces the Mirage μGraph generator, which automatically discovers potential μGraphs for an input tensor program. To generate μGraphs that capture optimizations at the kernel, block, and thread levels, Mirage must explore a significantly larger search space than existing superoptimizers, which only consider optimizations at the kernel level. Mirage employs two key techniques to address this challenge. First, based on the observation that optimizations at the kernel and block levels are substantially more critical to performance than optimizations at the thread level—since accessing device and shared memory is orders of magnitude more expensive than accessing register file—Mirage's μGraph generator employs a *hybrid approach*: it exhaustively considers all possible graphs up to a certain size at the kernel and block levels, and uses a rule-based strategy to construct graphs at the thread level. This approach reduces the search space while retaining

Algorithm 1 Mirage's hybrid μGraph generation algorithm.

Input: A LAX program with a computation graph G_{ref}
Output: A set of μGraphs $\mathcal{S}$

```
1:  E_O ← E(G_ref)
2:  S_0, S ← ∅
3:  GENERATENEXTKERNELOPERATOR(Inputs(G_ref))
4:  for all G ∈ S_0 do
5:    S ← S ∪ {THREADGRAPHCONSTRUCTION(G)}

6:  function GENERATENEXTKERNELOPERATOR(G_K)
7:    S_0 ← S_0 ∪ {G_K}
8:    for all kernel graph op type t; input set I do
9:      if rank(I,t) > rank(op.I, op.t) for each op ∈ G_K then
10:       if t is a pre-defined operator then
11:         if o := CONSTRUCTOP(G_K, I, t) is valid then
12:           GENERATENEXTKERNELOPERATOR(G_K ∪ {o})
13:       else                                  ▷ t is a graph-defined operator
14:         for all gridDims; forloopDims do
15:           G_B ← TBGraph(I, gridDimd, forloopDims)
16:           GENERATENEXTBLOCKOPERATOR(G_K, G_B)

17: function GENERATENEXTBLOCKOPERATOR(G_K, G_B)
18:   if all shared tensors in G_B are consumed then
19:     if o := CONSTRUCTOP(G_K, G_B.I, G_B) is valid then
20:       GENERATENEXTKERNELOPERATOR(G_K ∪ {o})
21:   for all block graph op type t; input set I do
22:     if rank(I,t) > rank(op.I, op.t) for each op ∈ G_B then
23:       if o := CONSTRUCTOP(G_B, I, t) is valid then
24:         GENERATENEXTBLOCKOPERATOR(G_K, G_B ∪ {o})

25: function CONSTRUCTOP(G, I, attrs)
26:   E ← EXPRINFR(E(I), attrs)                 ▷ Refer to Table 1
27:   if SUBEXPR(E, E_O) then                   ▷ Prune via abstract expressions
28:     S ← G.outputTensorShapeInfr(I, attrs)   ▷ Check tensor shape
29:     if S.valid, G.mAlloc + S.size ≤ G.mLimit then   ▷ Check memory
30:       return G.constructOp(I, attrs)
31:   return Invalid

32: function THREADGRAPHCONSTRUCTION(G)
33:   G_fused ← G
34:   while ∃o ∈ G_fused that can be fused with a preceding operator do
35:     G_fused ← FUSEOP(G_fused, o)
36:   return G_fused
```

most performance-critical optimizations. Second, to further prune the search space, Mirage introduces a pruning technique based on an abstraction of μGraphs called *abstract expression*, which reduces the number of μGraphs Mirage must consider while providing a certain theoretical guarantee on the optimality of the discovered μGraphs. We introduce the hybrid μGraph generation algorithm in §4.1 and §4.2, and the expression-guided pruning techniques in §4.3.

4.1 Kernel and Block Graph Generation

Mirage generates kernel and block graphs incrementally and leverages several pruning techniques to reduce the search space, as shown in the second part of Figure 5. Specifically, Mirage maintains a *prefix* of a valid μGraph and iteratively extends it with new operators. For a graph $G=(V,E)$ we say that $G'=(V',E')$ is a *prefix* of G if it is a subgraph of G such that $\forall u \in V', \forall (v,u) \in E, v \in V'$.

To generate the next operator in the kernel graph, Mirage enumerates the kernel operator type t and the input tensor set I. If t represents the graph-defined operator type, Mirage generates the associated block graph that defines its kernel computation by (1) enumerating the grid and for-loop dimensions (introduced in §2), which enables Mirage to calculate the input tensor shapes of the block graph; and (2) performing a nested generation procedure similar to that used at the kernel level but without considering graph-defined operators. Line 6-16 and line 17-24 in Algorithm 1 show how Mirage generates kernel and block operators, respectively. Mirage checks tensor shape (line 28) and memory usage (line 29) before adding an operator, ensuring a valid prefix.

To ensure that identical μGraphs are generated only once, Mirage defines the *canonical form* of μGraphs. Given a μGraph G with its operators in topological order $o_1, \ldots, o_n$, the *index* of the j-th output of o_i is defined as a tuple (i, j). Each operator o_i in G is assigned a *rank* $(input_i, type_i)$, where $input_i$ is the list of input tensor indices of o_i, and $type_i$ is the operator type. A μGraph is in canonical form if its operators are ordered in increasing rank. Mirage generates only μGraphs in canonical form by requiring that operators be added in increasing order of rank (line 9 and 22). This approach does not prune out any valid solutions, since each μGraph can be transformed to canonical form by reordering the operators.

In addition, Mirage utilizes the *abstract expression* technique to prune out prefixes that do not satisfy certain constraints, which will be introduced in §4.3.

4.2 Thread Graph Construction

While a similar nested generation strategy can be applied to thread graphs, Mirage instead constructs them using a transformation-based approach (see the third panel of Figure 5 and lines 4–5 in Algorithm 1) to reduce the search space. Mirage applies operator fusion when constructing thread graphs, which reduces access to shared memory by reusing tensors in register file whenever possible. For example, Mirage fuses the three element-wise operators (`Mul`, `Sqrt`, and `Div`) in Figure 3b into a thread graph, avoiding saving intermediate results to shared memory and keeping the entire computation of these operators in register file. While our current implementation focuses on operator fusion, additional rule-based transformations can be used to construct thread graphs.

4.3 Pruning via Abstract Expressions

When searching the space of possible μGraphs, we aim to avoid μGraph prefixes whose intermediate results cannot contribute to the desired computation. For example, for the input program $X \cdot Z + Y \cdot Z$, we can prune a prefix that computes $X \cdot Y$, but we should not prune one that computes $X + Y$, as

Table 1: Operators supported by Mirage. The second column shows the graph levels supporting each operator (K, B and T denote kernel, block, and thread graphs, respectively). The last column defines the abstract expressions for the outputs of each operator, where E maps tensors to their abstract expressions.

μGraph Operator	Graph Level	Abstract Expression of Output Tensor
InIter	B	$\mathrm{E}(\mathrm{InIter}(X)) = \mathrm{E}(X)$
OutSaver	B	$\mathrm{E}(\mathrm{OutSaver}(X)) = \mathrm{E}(X)$
Matmul	K, B, T	$\mathrm{E}(\mathrm{Matmul}(X,Y)) = \mathsf{sum}(k, \mathsf{mul}(\mathrm{E}(X), \mathrm{E}(Y)))$[1]
Sum	K, B, T	$\mathrm{E}(\mathrm{Sum}(d_r, k_r, X)) = \mathsf{sum}(k_r, \mathrm{E}(X))$[2]
EwAdd	K, B, T	$\mathrm{E}(\mathrm{EwAdd}(X,Y)) = \mathsf{add}(\mathrm{E}(X), \mathrm{E}(Y))$
EwMul	K, B, T	$\mathrm{E}(\mathrm{EwMul}(X,Y)) = \mathsf{mul}(\mathrm{E}(X), \mathrm{E}(Y))$
EwDiv	K, B, T	$\mathrm{E}(\mathrm{EwDiv}(X,Y)) = \mathsf{div}(\mathrm{E}(X), \mathrm{E}(Y))$
EwExp	K, B, T	$\mathrm{E}(\mathrm{EwExp}(X)) = \mathsf{exp}(\mathrm{E}(X))$
Repeat	K, B	$\mathrm{E}(\mathrm{Repeat}(X)) = \mathrm{E}(X)$
Reshape	K, B	$\mathrm{E}(\mathrm{Reshape}(X)) = \mathrm{E}(X)$
Sqr	K, B	$\mathrm{E}(\mathrm{Sqr}(X)) = \mathsf{mul}(\mathrm{E}(X), \mathrm{E}(X))$
Sqrt	K, B	$\mathrm{E}(\mathrm{Sqrt}(X)) = \mathsf{sqrt}(\mathrm{E}(X))$
SiLU	K, B	$\mathrm{E}(\mathrm{SiLU}(X)) = \mathsf{silu}(\mathrm{E}(X))$
Accum	B	$\mathrm{E}(\mathrm{Accum}(X,m,i)) = \mathsf{sum}(i, \mathrm{E}(X))$ if $m = \phi$ else $\mathrm{E}(X)$ [3]

[1] k means the size of the last dimension of A, i.e., the reduction dimension. Matmul is performed on the inner most two dimensions and leading dimensions are batched.
[2] Sum along the dimension d_r for every k_r elements.
[3] Accumulate the results of i for-loop iterations along fmap m.

$(X+Y)\cdot Z$ is equivalent to the input program. However, how can we determine whether a prefix can contribute to a desired computation while searching for that computation? Below, we develop a pruning technique driven by this intuition that circumvents the "chicken and egg" problem via *abstraction*. We first present the abstraction—*abstract expressions*—and then explain how it is used for pruning. Finally, we offer a theoretical guarantee that, under certain conditions, this pruning does not exclude the optimal μGraph.

Abstract expressions. Recall that an edge in a μGraph corresponds to a tensor-valued function of the input tensors. Intuitively, abstract expressions abstract these functions by ignoring the differences between elements of the same input tensor. Formally, abstract expressions are first-order logic terms over the theory of integers and uninterpreted functions. In a μGraph, the abstract expression of each edge, denoted by $\mathrm{E}(\cdot)$, is defined in Table 1. When computing a μGraph's abstract expression, all graph-defined operators are "inlined". Specifically, the expressions computed for a graph-defined operator's inputs are passed into its lower-level graph, and the resulting output expressions of that lower-level graph become the output expressions of the graph-defined operator. Figure 6 shows the abstract expressions for a subgraph of attention.

While abstract expressions capture some information about the function computed at each edge, they also abstract away many details. For example, if X is a $k \times k$ matrix, summing over the rows and summing over the columns both yield the same abstract expression—$\mathsf{sum}(k, \mathrm{E}(X))$. But keeping k as part of the abstract expression is crucial for effective pruning.

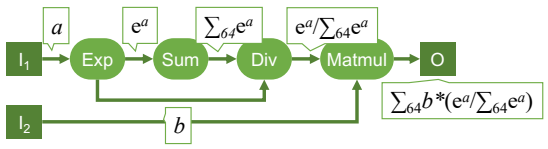


Figure 6: Illustration of abstract expressions. The abstract expressions of tensors are annotated on edges. A human-friendly notation is used here: e^a denotes $\mathsf{exp}(a)$, $\sum_k a$ denotes $\mathsf{sum}(k,a)$, a/b denotes $\mathsf{div}(a,b)$, and $a*b$ denotes $\mathsf{mul}(a,b)$. The tensors I_1, I_2 and O are all 64×64 matrices.

Abstract subexpression and pruning. We use abstract expressions to prune the search space of μGraphs by formalizing two relations over abstract expressions: equivalence and abstract subexpression. Specifically, we prune any μGraph prefix whose abstract expression is not a subexpression of some abstract expression equivalent to that of the input program. We formalize abstract expressions as uninterpreted functions in first-order logic over the theory of integer arithmetic and uninterpreted functions, and use an SMT solver to reason about them based on two sets of axioms in Table 2: A_{eq} and A_{sub}.

First, A_{eq} axiomatizes equivalence between abstract expressions. As will become clear below, these axioms need not be sound—it is not required that μGraphs with equivalent abstract expressions are functionally equivalent, since non-equivalent μGraphs can have the same abstract expression. Second, A_{sub} axiomatizes the subexpression relation between abstract expressions. A key property of A_{sub} is that whenever a μGraph G_1 is a prefix of G_2—meaning G_2 can be constructed by extending G_1 with additional operators—$\mathrm{E}(G_1)$ is an abstract subexpression of $\mathrm{E}(G_2)$; formally, $A_{\mathrm{sub}} \models \mathsf{subexpr}(\mathrm{E}(G_1), \mathrm{E}(G_2))$, where $\models$ denotes entailment modulo the theory of integer arithmetic and uninterpreted functions.

During the search, Algorithm 1 first computes the abstract expression of the input LAX program, denoted E_O, and prunes any μGraph prefix G if $A_{\mathrm{eq}} \cup A_{\mathrm{sub}} \not\models \mathsf{subexpr}(\mathrm{E}(G), E_O)$. That is, a graph is pruned if its abstract expression is not a subexpression of E_O. This check is performed using an SMT solver (Z3 [18]). As an optimization, the results of these checks are cached and reused, since Mirage may encounter multiple μGraphs with identical abstract expressions during the search.

Theoretical guarantee and the pruning-optimality tradeoff. Intuitively, our pruning would keep any prefix that can lead to a μGraph whose abstract expression is equivalent (according to A_{eq}) to that of the input LAX program. Formally:

Theorem 1 (Pruning via Abstract Expressions)**.** For an input μGraph G_0, and a μGraph G equivalent to G_0, if $A_{\mathrm{eq}} \models E(G_0) = E(G)$ then G will be generated by Algorithm 1.

Proof. By Tables 1 and 2, we show that for any operator op, if $Y = op(X_1, \ldots, X_n)$, then $A_{\mathrm{sub}} \models \mathsf{subexpr}(\mathrm{E}(X_i), \mathrm{E}(Y))$ for $1 \le i \le n$. That is, the abstract expression of each input to op

Table 2: Axiomatization of abstract expressions used for pruning. Mirage checks whether an abstract expression E_1 is a subexpression of E_2 by querying an SMT solver to check if $\mathsf{subexpr}(E_1, E_2)$ is entailed by these axioms. All variables in these axioms are universally quantified.

Abstract Expression Property	Comment
Equivalence Axioms A_{eq}	
$\forall x,y.\ \mathsf{add}(x,y)=\mathsf{add}(y,x)$	commutativity
$\forall x,y.\ \mathsf{mul}(x,y)=\mathsf{mul}(y,x)$	commutativity
$\forall x,y,z.\ \mathsf{add}(x,\mathsf{add}(y,z))=\mathsf{add}(\mathsf{add}(x,y),z)$	associativity
$\forall x,y,z.\ \mathsf{mul}(x,\mathsf{mul}(y,z))=\mathsf{mul}(\mathsf{mul}(x,y),z)$	associativity
$\forall x,y,z.\ \mathsf{add}(\mathsf{mul}(x,z),\mathsf{mul}(y,z))=\mathsf{mul}(\mathsf{add}(x,y),z)$	distributivity
$\forall x,y,z.\ \mathsf{add}(\mathsf{div}(x,z),\mathsf{div}(y,z))=\mathsf{div}(\mathsf{add}(x,y),z)$	associativity
$\forall x,y,z.\ \mathsf{mul}(x,\mathsf{div}(y,z))=\mathsf{div}(\mathsf{mul}(x,y),z)$	associativity
$\forall x,y,z.\ \mathsf{div}(\mathsf{div}(x,y),z)=\mathsf{div}(x,\mathsf{mul}(y,z))$	associativity
$\forall x.\ x=\mathsf{sum}(1,x)$	identity reduction
$\forall x,i,j.\ \mathsf{sum}(i,\mathsf{sum}(j,x))=\mathsf{sum}(i*j,x)$	associativity
$\forall x,y,i.\ \mathsf{sum}(i,\mathsf{add}(x,y))=\mathsf{add}(\mathsf{sum}(i,x),\mathsf{sum}(i,y))$	associativity
$\forall x,y,i.\ \mathsf{sum}(i,\mathsf{mul}(x,y))=\mathsf{mul}(\mathsf{sum}(i,x),y)$	distributivity
$\forall x,y,i.\ \mathsf{sum}(i,\mathsf{div}(x,y))=\mathsf{div}(\mathsf{sum}(i,x),y)$	distributivity
$\forall x,y.\ \mathsf{mul}(\mathsf{exp}(x),\mathsf{exp}(y))=\mathsf{exp}(\mathsf{add}(x,y))$	distributivity
$\forall x,y.\ \mathsf{mul}(\mathsf{sqrt}(x),\mathsf{sqrt}(y))=\mathsf{sqrt}(\mathsf{mul}(x,y))$	distributivity
Subexpression Axioms A_{sub}	
$\forall x,y.\ \mathsf{subexpr}(x,\mathsf{add}(x,y))$	
$\forall x,y.\ \mathsf{subexpr}(x,\mathsf{mul}(x,y))$	
$\forall x,y.\ \mathsf{subexpr}(x,\mathsf{div}(x,y))$	
$\forall x,y.\ \mathsf{subexpr}(y,\mathsf{div}(x,y))$	
$\forall x.\ \mathsf{subexpr}(x,\mathsf{exp}(x))$	
$\forall x.\ \mathsf{subexpr}(x,\mathsf{sqrt}(x))$	
$\forall x.\ \mathsf{subexpr}(x,\mathsf{silu}(x))$	
$\forall x,i.\ \mathsf{subexpr}(x,\mathsf{sum}(i,x))$	
$\forall x.\ \mathsf{subexpr}(x,x)$	reflexivity
$\forall x,y,z.\ \mathsf{subexpr}(x,y)\wedge\mathsf{subexpr}(y,z)\rightarrow\mathsf{subexpr}(x,z)$	transitivity

is always a subexpression of *op*'s output. Given that A_{sub} includes reflexivity and transitivity axioms, it follows that for any G' that is a prefix of G, $A_{\text{sub}} \models \mathsf{subexpr}(\mathrm{E}(G'),\mathrm{E}(G))$. Together with the assumption that $A_{\text{eq}} \models E(G_0)=E(G)$, we have $A_{\text{eq}} \cup A_{\text{sub}} \models \mathsf{subexpr}(\mathrm{E}(G'),\mathrm{E}(G_0))$. Thus, no prefix of G will be pruned, and Mirage will generate G. □

The theorem highlights the role of abstract expressions in solving the "chicken and egg" problem outlined above. To decide if a prefix μGraph is useful, we reason about whether it is a prefix of a useful computation *in the abstract*. The choice of abstraction and the axioms A_{eq} represents a tradeoff between optimality and pruning. As Theorem 1 shows, we are only guaranteed to find the optimal μGraph whose abstract expression is equivalent to that of the input program under A_{eq}. Stronger axioms expand the set of μGraphs covered by the theorem but reduce pruning effectiveness, since more prefixes would pass the subexpression test. In particular, note that A_{eq} does not include cancellation rules (e.g., $\mathsf{div}(\mathsf{mul}(x,y),y)=y$). As a result, Mirage may miss some equivalent μGraphs. However, including such axioms would make everything a subexpression of everything, therefore nulling desired pruning. As our evaluation shows, the chosen A_{eq} yields a good balance between pruning and optimality.

5 Probabilistic Equivalence Verifier

Mirage's *probabilistic equivalence verifier* checks if a candidate μGraph is equivalent to the desired LAX program. The key idea is to evaluate both on *random inputs* in two finite fields. Using finite fields instead of floating point numbers not only avoids floating point errors but also provides a strong theoretical guarantee: the probability of accepting a non-equivalent μGraph can be made arbitrarily low.

For general programs, random testing can hardly provide any correctness guarantee. However, we show that for LAX programs (formally defined below), random testing offers a probabilistic correctness guarantee, and repeated tests can reduce the error probability to an arbitrarily small threshold.

Prior work [46] has applied a similar technique to check equivalence between tensor programs that contain only linear operators (e.g., matrix multiplication, convolution). We develop a random testing technique that also supports division and exponentiation, which are needed for many DNN optimizations (e.g., the RMSNorm example in §3).

Mirage verifies equivalence between LAX μGraphs (linear, division, and an exponentiation) defined below. We introduce the main theoretical results in §5.1 and present Mirage's verification methodology in §5.2.

Definition 5.1 (LAX μGraph)**.** A μGraph G is a LAX μGraph if (1) G contains only multi-linear operators[4], division, and exponentiation, and (2) every path from an input to an output in G includes at most one exponentiation.

5.1 Theoretical Foundations

Without loss of generality, we assume a LAX μGraph G takes n input tensors and produces one output tensor. Our theoretical results directly generalize to LAX μGraph with multiple outputs. Since each LAX μGraph includes linear operators, divisions, and at most one exponentiation along each path, the computation for each entry of the output tensor can be expressed in the following form (by using standard identities such as $\frac{\frac{a}{b}}{\frac{c}{d}}=\frac{ad}{bc}$, $\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd}$, $e^x e^y = e^{x+y}$):

$$\frac{\sum_{i=1}^{k} f_i \exp(g_i/h_i)}{\sum_{i=1}^{k'} f'_i \exp(g'_i/h'_i)} \tag{2}$$

where f_i, g_i, h_i, f'_j, g'_j and h'_j ($1 \le i \le k$, $1 \le j \le k'$) are polynomials over the entries of the input tensors.

The main theoretical result that underpins our randomized equivalence verification is the following theorem, which extends polynomial identity testing (PIT) [37, 54] on finite fields to LAX μGraphs. Note that the difference of two LAX

[4]Operator *op* with n inputs is multi-linear if *op* is linear to all inputs I_k:
(1) $\forall X,Y.op(I_1,...,I_{k-1},X,I_{k+1},...,I_n) + op(I_1,...,I_{k-1},Y,I_{k+1},...,I_n) = op(I_1,...,I_{k-1},X+Y,I_{k+1},...,I_n)$, and
(2) $\alpha \cdot op(I_1,...,I_{k-1},X,I_{k+1},...,I_n) = op(I_1,...,I_{k-1},\alpha\cdot X,I_{k+1},...,I_n)$.

Table 3: Arithmetic operations for random testing. Mirage selects two prime numbers p and q such that q divides $p-1$. x_p and x_q are values from the finite fields $\mathbb{Z}_p$ and $\mathbb{Z}_q$, respectively. The notation x^{-1} and $\sqrt{x}$ represents the multiplicative inverse and square root of x in the corresponding finite field. Specifically, $xx^{-1} \bmod p = 1$ and $\sqrt{x}\sqrt{x} \bmod p = x$.

Opt.	Opd. 1	Opd. 2	Output
Add.	(x_p, x_q)	(y_p, y_q)	$\big((x_p + y_p) \bmod p, (x_q + y_q) \bmod q\big)$
Sub.	(x_p, x_q)	(y_p, y_q)	$\big((x_p - y_p) \bmod p, (x_q - y_q) \bmod q\big)$
Mul.	(x_p, x_q)	(y_p, y_q)	$\big(x_p y_p \bmod p, x_q y_q \bmod q\big)$
Div.	(x_p, x_q)	(y_p, y_q)	$\big(x_p y_p^{-1} \bmod p, x_q y_q^{-1} \bmod q\big)$
Exp.	(x_p, x_q)	–	$\big(\omega^{x_q} \bmod p, -\big)$
Sqrt.	(x_p, x_q)	–	$\big(\sqrt{x_p}, \sqrt{x_q}\big)$

μGraphs is also of the form of Equation (2). Therefore, identity testing of two LAX μGraphs reduces to testing if an expression of that form is zero. Due to the presence of exponentiation, we use two finite fields instead of one.[5]

Theorem 2. Let P be a function of the form described in Equation (2), where $f_i, g_i, h_i, f_i', g_i', h_i'$ are non-zero polynomials of degree at most d with integer coefficients between $[-w, w]$. Let p, q be primes such that $q \mid p-1$ and $q > 2w$. Let $\mathcal{G}$ be the set of q-th roots of unity in $\mathbb{Z}_p$. If P is not a zero function, then [27]

$$\Pr_{(\vec{u},\vec{v},\omega) \leftarrow \mathbb{Z}_p^N \times \mathbb{Z}_q^N \times \mathcal{G}} \left[\frac{\sum_{i=1}^{k} f_i(\vec{u}) \omega^{g_i(\vec{v})/h_i(\vec{v})}}{\sum_{i=1}^{k'} f_i'(\vec{u}) \omega^{g_i'(\vec{v})/h_i'(\vec{v})}} \right] \leq 8dk^4/q + q^{-1/k^2}.$$

5.2 Random Tests over Finite Fields

Mirage leverages Theorem 2 to probabilistically verify the equivalence of two μGraphs by performing random testing over the finite fields $\mathbb{Z}_p$ and $\mathbb{Z}_q$ as defined in Theorem 2. To check the equivalence of two μGraphs, Mirage first generates input tensors, with each entry uniformly sampled from $\mathbb{Z}_p \times \mathbb{Z}_q$. Mirage also samples ω uniformly from the set of q-roots of unity in $\mathbb{Z}_p$, which is used for exponentiation. Mirage then evaluates the two μGraphs on these inputs using the operations defined in Table 3. As explained in §5.1, $\mathbb{Z}_p$ and $\mathbb{Z}_q$ are used for computations outside and inside the exponent, respectively. All operations except exponentiation are implemented via modular arithmetic in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ independently. For exponentiation, Mirage uses the value x_q from $\mathbb{Z}_q$ and computes $\omega^{x_q} \bmod p$ to obtain a result in $\mathbb{Z}_p$.

Note that in a LAX μGraph, exponentiation is performed at most once along each path. Finally, Mirage checks whether the two μGraphs produce identical outputs. This process is repeated multiple times, and the two μGraphs are considered equivalent if they pass all random tests. The following theorem, which follows from Theorem 2, shows that this process can yield an arbitrarily low error rate.

Theorem 3. Equivalent μGraphs always pass μGraph verification. For two non-equivalent μGraphs and a given probability threshold $0 < \delta \leq 1$, the μGraphs pass all $\Omega(\frac{k^2}{\ln q} \cdot \ln \frac{1}{\delta})$ random tests with probability at most δ.

Numerical stability. While the theorem bridges finite fields and real-number computations, discrepancies can arise between real-number computations and floating-point operations, particularly involving overflow or underflow due to large intermediate values. Mirage employs floating-point tests to filter out μGraphs with significant numerical errors.

6 μGraph Optimizer

For each verified μGraph, Mirage's *μGraph optimizer* maximizes its performance by further performing *layout optimization*, *operator scheduling*, and *memory planning*, as shown in Figure 1. Mirage defers these μGraph optimizations until after verification for two reasons. First, these optimizations *do not* affect the correctness of the generated μGraphs; omitting them when generating μGraphs reduces the search space Mirage must consider, since μGraphs with the same graph topology but different choices of tensor layouts, operator orders, or memory allocation plans are considered identical by the μGraph generator. Second, applying these optimizations after verification also reduces the search space for these optimizations, since the μGraph optimizer only needs to optimize μGraphs that are functionally equivalent to the input.

Tensor layouts. The μGraph optimizer explores possible data layouts for all intermediate tensors at the kernel, block, and thread levels and chooses the best combinations to maximize performance. We formulate layout selection as a constrained optimization problem and solve it optimally using an integer linear programming (ILP) algorithm. Specifically, for each tensor t and each possible layout l for t, we introduce a boolean variable $B_{t,l}$ to indicate whether tensor t uses layout l. Operators at the kernel, block, and thread levels may impose various constraints on tensor layouts. For example, to use kernels from the cuBLAS library [16] for matrix multiplication, the innermost dimension of the two input tensors must be among the last two dimensions. These restrictions are converted into a series of linear constraints on $B_{t,l}$. Different tensor layouts may lead to varying performance. For example, some input tensor layouts support bulk copies from device to shared memory, while others do not. Mirage introduces a cost function to model the performance of each operator under different layout choices. Mirage uses an off-the-shelf ILP solver (i.e., Z3 [18]) to find an optimal layout strategy that satisfies all layout constraints while minimizing cost.

[5] We use two primes p and q for polynomial identity testing [37, 54] outside and inside the exponents, respectively. The condition q divides $p-1$ is to ensure the existence of q-th roots of unity in $\mathbb{Z}_p$.

Table 4: DNN benchmarks used in our evaluation.

Name	Description	Base Architecture
GQA	Group-query attention	LLaMA-3-70B [41]
QKNorm	QK normalization with attention	Chameleon-7B [40]
RMSNorm	RMS normalization with linear	LLaMA-2-7B [44]
LoRA	Low-rank adaptation	GPT-3-7B-LoRA [6]
GatedMLP	Gated multi-layer perceptron	Falcon-7B [10]
nTrans	Normalized Tarnsformer	nGPT-1B [28]

Operator scheduling. In a μGraph, there are multiple topological orders to execute operators, and different orders may yield different performance. For a given input μGraph, the μGraph optimizer identifies an efficient strategy to schedule operators by minimizing thread-level synchronization within each thread block (i.e., `__syncthreads()` in CUDA). To achieve this goal, Mirage labels each node with a *depth*, defined as the length of the longest path from any input operator to that node. Mirage uses a dynamic programming algorithm to compute the depth of each node and schedules all operators in ascending order of their depths. This approach minimizes the number of thread-level synchronizations required in the generated CUDA kernel, as Mirage only needs to insert synchronization points between operators with different depths.

Memory planning. A third class of post-verification optimizations is memory planning, which determines memory offsets for all intermediate tensors at the kernel, block, and thread levels. Mirage formulates memory planning as a *dynamic storage allocation* problem and exhaustively enumerates all possible allocation plans to discover an optimal strategy.

7 Implementation

Mirage is implemented in 30K lines of code in C++, CUDA, and Python. Kernel operators are implemented with the cuDNN and cuBLAS libraries [15, 16], and block and thread operators are implemented using cuTLASS [2] and CUDA PTX. For each input tensor program, Mirage automatically generates and verifies potential μGraphs. For each verified μGraph, Mirage produces CUDA source code for all custom kernels of the μGraph and compiles the code into binary using the CUDA compiler. This approach enables just-in-time (JIT) compilation and deployment for general tensor programs, and the generated kernels can be directly integrated into a PyTorch program with a few lines of code changes. Mirage's SMT and ILP solvers are implemented using Z3 4.12.6 [18].

Our implementation supports the operators listed in Table 1. Mirage can be extended to include new operators, such as variants of convolution or matrix multiplication, at the kernel, block, and/or thread levels. To support a new linear operator, Mirage requires (1) a float-pointing implementation of the operator at the kernel, block, and/or thread levels, which is used by the μGraph optimizer to generate CUDA kernels; (2) an implementation of the operator over modular arithmetic (see §5); and (3) an extension to the abstract expression axioms A_{eq} and A_{sub} for the operator (see §4.3).

To utilize Theorems 2 and 3, random tests should be performed with sufficiently large prime numbers p and q and iterated multiple times. Our current implementation uses the largest values of p and q whose product fits in 16-bit integers (i.e., $p=227, q=113$) to run these random tests on GPUs. We leverage Mirage's GPU optimizations–such as keeping intermediate results in shared memory–to accelerate the search procedure. We also perform a single random test without iterating it and compare all elements of the output tensors. We note that this equivalence verification procedure does not introduce false negatives. While it could, in theory, introduce false positives, we have not observed any in practice. For these reasons, we consider this procedure sufficient for the search process and plan to add a final verification step that provides the theoretical guarantees only for the best μGraph at the end of the optimization process.

Equivalence verification for non-LAX programs. While Mirage can generate μGraphs for arbitrary tensor programs, the probabilistic equivalence verifier is limited to LAX programs and does not support certain DNN operators such as ReLU [32]. As an alternative, we have developed a solver-based verifier for arbitrary tensor programs. The verifier relies on user-provided mathematical properties of individual operators (e.g., linearity, associativity, commutativity, and distributivity) defined in first-order logic and uses these properties to verify equivalence using an automated theorem prover. Compared to the probabilistic equivalence verifier, the solver-based verifier supports more general programs, while requiring additional manual effort to specify the properties of each new operator. A detailed discussion of the solver-based verifier is beyond the scope of this paper.

8 Evaluation

8.1 Experimental Setup

Since Mirage is a superoptimizer for LAX programs, we focus our evaluation on various DNN benchmarks commonly used in existing DNNs, each of which is a LAX program. These benchmarks provide the most fine-grained way to compare the performance of Mirage and existing systems. Table 4 shows the six benchmarks in our evaluation. GQA, RMSNorm, and GatedMLP are the main building blocks of large language models (LLMs). QKNorm introduces query-key normalization before attention to enhance model convergence [40]. LoRA enables low-rank adaptation for fine-tuning a DNN on different tasks. We use a context length of 8K for GQA and 4K for QKNorm, corresponding to the maximum supported by LLaMA-3-70B [41] and Chameleon-7B [40], respectively. In addition, we also evaluate how Mirage-generated kernels

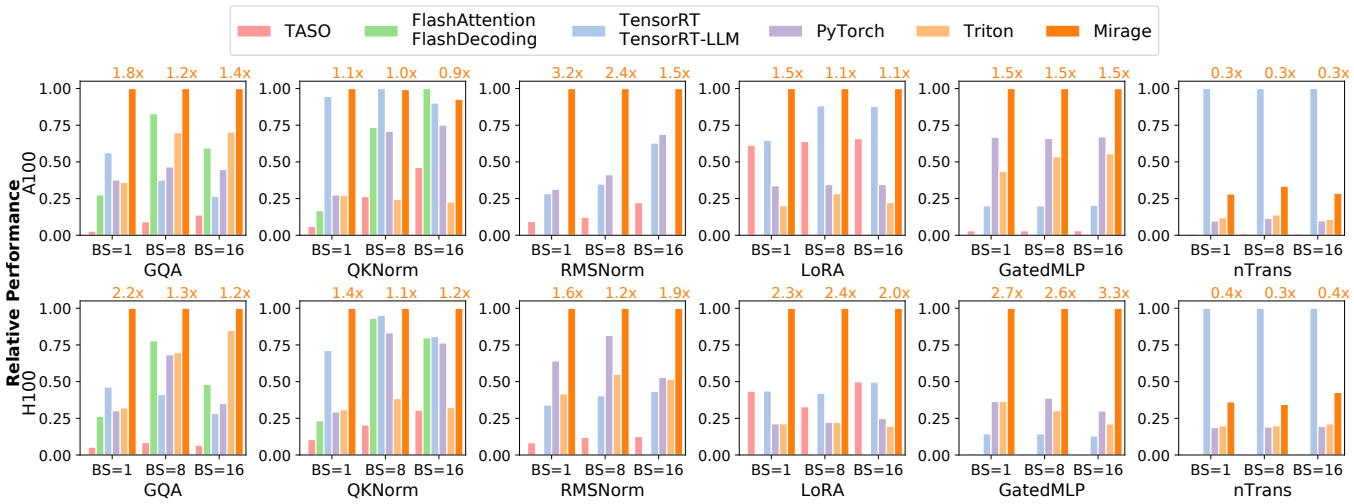


Figure 7: Comparing Mirage with existing systems for 6 benchmarks on an A100 and an H100 GPU. The performance of all systems are normalized by Mirage (higher is better). Numbers above the Mirage bars show the speedup over the best baselines.

improve the end-to-end performance of full DNNs, including Chameleon [40], nGPT [28], LLaMA-3 [41], and LoRA [22].

The experiments were conducted on NVIDIA A100 and H100 GPUs, each with 40GB of memory. All our benchmarks fit on a single GPU except GQA (used for LLaMA-2-70B), which is generally parallelized across four GPUs using tensor model parallelism [39]. Therefore, we evaluate GQA under this parallelism strategy, where the eight key-value heads are equally partitioned across four GPUs. Since the performance of Mirage and all baselines depends only on the shapes of the input tensors, we repeat each experiment 1,000 times using random inputs and report the average run time.

One of our benchmarks, LoRA, requires concatenation to express a common optimization: fusing two matrix multiplications via concatenation. To support this optimization in Mirage, we introduce a new linear operator that takes four inputs and computes $f(W,X,Y,Z)=(W\|X)\times(Y\|Z)$, where $\|$ is tensor concatenation. This operator is equivalent to computing $W\times Y+X\times Z$. We define the abstract expression associated with this operator as: $\mathrm{E}(f(W,X,Y,Z))=\mathsf{add}(\mathsf{sum}(k_1,\mathsf{mul}(\mathrm{E}(W),\mathrm{E}(Y))),\mathsf{sum}(k_2,\mathsf{mul}(\mathrm{E}(X),\mathrm{E}(Z))))$, where k_1 and k_2 are the last dimensions of W and X.

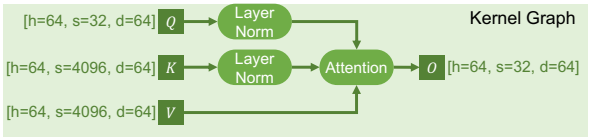
Unless otherwise stated, Mirage considers up to 5 operators in the kernel graph and up to 11 operators in each block graph.

8.2 Benchmark Results

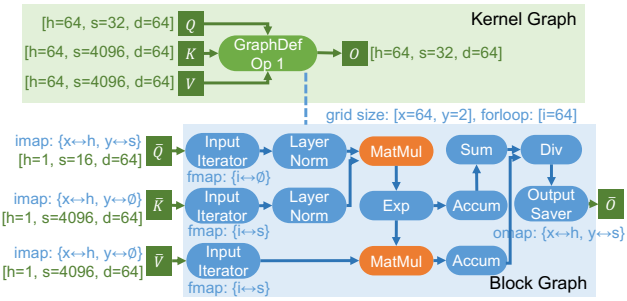
Figure 7 compares the performance of Mirage with systems on six DNN benchmarks on NVIDIA A100 and H100 GPUs. All systems use half-precision floating points to run these DNN benchmarks. TASO [25] and PET [46] are DNN superoptimizers that automatically generate algebraic transformations at the kernel level. We report a combined TASO/PET baseline, as the latest TASO implementation includes PET's partially equivalent transformations as special substitutions. PyTorch [34] uses the highly optimized cuDNN and cuBLAS libraries [15, 16] to perform DNN operators on GPUs. For the PyTorch baseline, we enable `torch.compile` and use FlashAttention kernels to maximize performance. TensorRT and its LLM variant TensorRT-LLM include a set of manually designed and highly optimized kernels for common tensor operators such as attention [42]. FlashAttention and its inference variant FlashDecoding are manually written kernels for efficient attention [17, 21]. Finally, Triton is a schedule-based optimizer to generate high-performance kernels and has been adopted in production systems, outperforming other schedule-based approaches [43]. All baselines use CUDA Graphs to minimize kernel launch overhead.

Compared to the best existing approaches, Mirage improves the performance of these benchmarks by up to 3.3× by combing algebraic transformations, schedule transformations, and the generation of new custom kernels. §3 shows the best discovered μGraphs for RMSNorm. Next, we present a case study for the remaining benchmarks.

GQA. Group-query attention is the backbone of LLMs and has been heavily optimized by existing frameworks. For example, FlashAttention and FlashDecoding are expert-designed attention kernels and have been adopted in existing LLM inference systems [17]. Mirage discovers these expert-designed kernels as well as other μGraphs that outperform them by up to 2.2×. The speedup is achieved by two additional optimizations on top of existing hand-written kernels. First, current approaches rely on fixed heuristics to determine the grid dimensions for GQA, which are suboptimal in certain scenarios. For example, TensorRT-LLM launches the GQA kernel with grid dimensions of (8, 2, 1) and (8, 2, 8) when the batch sizes are 1 and 8, respectively. However, both configurations can-



(a) The kernel graph for QKNorm and attention in existing systems.



(b) The best μGraph discovered by Mirage for QKNorm and attention.

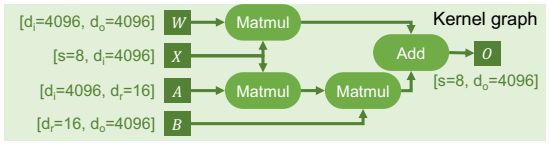
Figure 8: Comparing the μGraphs used by existing optimizers and Mirage for QKNorm and attention.

not fully utilize all SMs on A100 (108 SMs) and H100 (132 SMs) GPUs. In contrast, Mirage automatically searches for the best grid dimensions for each μGraph, resulting in full SM utilization. Further ablation study shows that the performance of the best μGraph discovered by Mirage degrades by 18% when using the same grid dimensions as TensorRT-LLM.

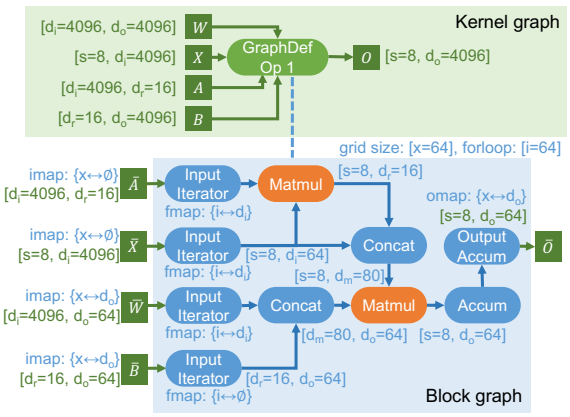
Second, existing approaches use fixed tensor dimensions to parallelize GQA across thread blocks. For example, FlashAttention [17] parallelizes attention across the *sample*, *head*, and *query sequence* dimensions, while FlashDecoding and TensorRT-LLM leverage the *sample*, *head*, and *key-value sequence* dimensions. Both strategies are efficient for conventional multi-head attention with many heads but suboptimal for GQA with fewer attention heads. In contrast, Mirage automatically selects the most efficient parallelization strategy by choosing among the sample, KV heads, query sequence, and key-value sequence dimensions. Moreover, Mirage generates different μGraphs tailored to different attention scenarios, reducing device memory access by up to 7× compared to the heuristics used in existing systems.

Implementing Mirage's μGraphs in existing systems is possible but requires extensive engineering effort to support different kernels for different scenarios. In contrast, Mirage automatically generates them and verify their correctness.

QKNorm. To reduce model divergence, several recent DNNs introduce query-key normalization (QKNorm) into the Transformer architecture [40]. QKNorm applies layer normalization to the query and key vectors before attention, as shown in Figure 8a. These additional normalization layers are not yet supported by existing attention implementations (e.g., FlashAttention and TensorRT-LLM) and require launching separate kernels for normalization and attention.



(a) The kernel graph for LoRA in existing systems.



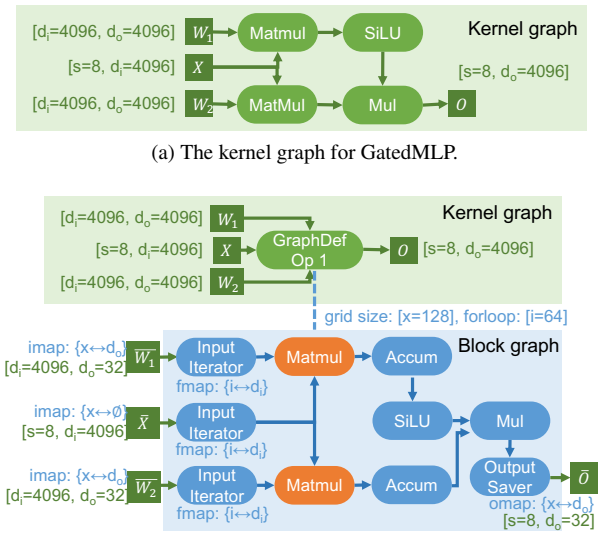
(b) The best μGraph discovered by Mirage for LoRA.

Figure 9: Comparing the tensor programs used by existing optimizers and by Mirage for LoRA: $O = W \times X + B \times A \times X$. Note that both matrices A and B are low-rank.

Mirage automatically discovers a μGraph that integrates QKNorm and attention computation into a custom kernel, as shown in Figure 8b. The μGraph reorganizes the attention computation to enable fusion with the two layer normalizations, which avoids writing intermediate results to GPU device memory and reduces the kernel execution time by up to 1.4×.

LoRA. Low-rank adaptation (LoRA) introduces a pair of low-rank adapters to the linear operators of a pre-trained DNN to improve its performance for downstream tasks. Existing tensor program optimizers launch separate kernels for the original linear operator and the two additional linear operators introduced by LoRA (Figure 9a), which introduces high kernel launch overheads since these LoRA operators involve minimal computation. Figure 9b shows the best μGraph discovered by Mirage for LoRA, which fuses the three `Matmul`s and the subsequent `Add` into a single kernel. Mirage reorganizes the computation into two block-level `Matmul`s by leveraging the following algebraic transformation: $W \times X + B \times A \times X = (W \| B) \times \big(X \| (A \times X)\big)$. The `Concat`s in Figure 9b do not involve any computation and are performed by updating tensor offsets in GPU shared memory. This μGraph reduces the execution cost of LoRA by 1.1-2.4×.

GatedMLP. Gated multi-layer perceptrons are commonly used in DNNs to capture non-linear representations. We use the GatedMLP configuration introduced in Falcon-7B [10], whose kernel graph is shown in Figure 10a. Existing tensor program optimizers generally fuse the two `Matmul`s in



(a) The kernel graph for GatedMLP.

(b) The best μGraph discovered by Mirage for GatedMLP.

Figure 10: Comparing the μGraphs used by existing optimizers and Mirage for GatedMLP.

a single kernel to reduce GPU device memory access, since the input tensor X only needs to be loaded once. However, this approach still requires launching multiple kernels and storing intermediate results—specifically, the output of the two `Matmul`s—in device memory, as the `SiLU` activation and elementwise multiplication are not fused with the `Matmul`s.

In contrast, the best μGraph discovered by Mirage (Figure 10b) performs the two `Matmul`s in parallel within the same block graph and fuses the remaining computation (i.e., `SiLU` and `Mul`) as post-processing steps within the same block graph. This approach yields 1.5× speedups on A100 GPUs and 2.7-3.3× speedups on H100 GPUs.

nTrans. To accelerate model training, nGPT introduces normalized Transformer, which normalizes all intermediate results in Transformer [28]. Formally, the computation is defined as $y = \texttt{Norm}(x + \alpha(\texttt{Norm}(h - x)))$, where `Norm` is a normalization layer, and x, h, and α are input tensors. Existing systems launch three separate kernels for nTrans, since it interleaves normalization and elementwise addition and multiplication. Mirage automatically discovers a μGraph that fuses the computation into a single kernel and stores all intermediate results in GPU shared memory. Mirage outperforms other baselines but is slower than TensorRT. This performance gap is because Mirage loads data from global memory to shared memory and writes it back for each tensor in graph-defined kernels. This design improves memory efficiency and enables asynchronous pipelines. However, for kernels with light computation, the overhead of these memory transfers can dominate the kernel runtime. To mitigate this overhead, we plan to extend Mirage to support bypassing shared memory during data loading, therefore avoiding unnecessary data movement.

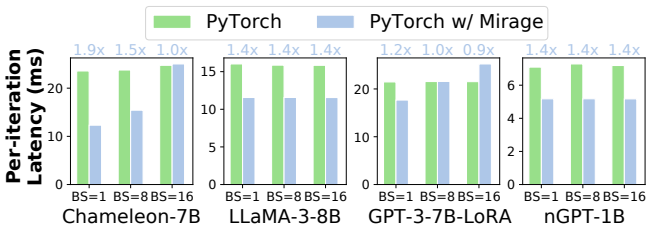


Figure 11: Comparing the end-to-end inference performance of PyTorch and PyTorch with Mirage-generated kernels.

Table 5: Ablation study on Mirage's techniques to accelerate μGraph generation. We evaluate the impact of multi-threading and abstract expressions on search time for RMSNorm.

Max # Ops in a Block Graph	Mirage	Mirage w/o Multithreading	Mirage w/o Abstract Expression
5	11 sec	58 sec	768 sec
6	16 sec	93 sec	19934 sec
7	22 sec	150 sec	> 10 h
8	24 sec	152 sec	> 10 h
9	26 sec	166 sec	> 10 h
10	26 sec	166 sec	> 10 h
11	28 sec	183 sec	> 10 h

8.3 End-to-end Results

In addition to the microbenchmark performance, we also evaluate how Mirage-generated kernels impact the end-to-end latency of commonly used DNNs. Mirage supports just-in-time compilation and deployment, and its generated kernels can be directly integrated into PyTorch programs. We compare PyTorch with its native handwritten CUDA kernels and PyTorch with Mirage-generated kernels on four DNN models. Figure 11 shows the results. Mirage reduces the end-to-end latency of these models by 0.9-1.9× by automatically generating highly optimized kernels. The improvement is achieved with a few lines of code changes to the PyTorch programs.

8.4 Search Time

In our evaluation, Mirage takes up to 4 hours to optimize a LAX program. This optimization is a one-time cost before deployment on the target hardware. This subsection provides detailed results and an ablation study of Mirage's search procedure, focusing on how its techniques enable the exploration of large μGraphs while maintaining low search time. In particular, we evaluate the impact of two techniques: pruning via abstract expressions (§4.3) and multi-threading. Table 5 reports the search times for RMSNorm as we vary the maximum number of operators allowed in a block graph.

Multi-threading significantly reduces the search time, while pruning via abstract expressions is crucial for the scalability of Mirage. Specifically, the pruning techniques allow Mirage to explore μGraphs whose block graphs can each have at most 11 operators, while disabling abstract expression pruning

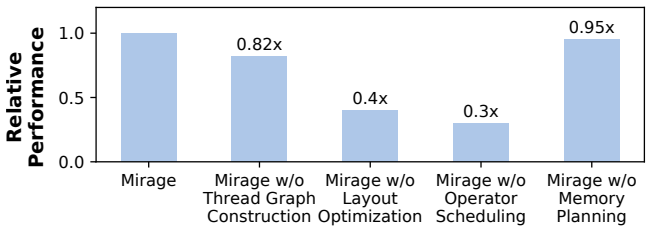


Figure 12: Ablation study on optimizations used in Mirage. We evaluate the performance degradation when disabling each optimization independently. The evaluation is performed on A100 for GQA with batch size 1.

restricts Mirage to handle block graphs with up to 6 operators within a 10-hour search window. Note that discovering the optimized μGraph for RMSNorm shown in Figure 3 requires exploring block graphs with 11 operators.

8.5 Ablation Study on Optimizations

We conduct an ablation study to evaluate the impact of thread graph construction and optimizations introduced in §6, including layout optimization, operator scheduling, and memory planning. Specifically, we measure the performance degradation of the best μGraph discovered by Mirage when each optimization is disabled independently. The study is conducted on an A100 using the GQA benchmark with a batch size of 1. The results, shown in Figure 12, indicate that disabling any individual optimization leads to a performance degradation ranging from 5% to 70%.

9 Related Work

Manually-designed kernels. Many existing frameworks, such as TensorFlow XLA [1, 9], PyTorch [34], and TensorRT [42], rely on GPU experts to manually design kernels for ML operators. Recently, significant engineering effort has been dedicated to hand-optimizing GPU kernels for commonly used DNNs, particularly foundation models [12]. For example, to accelerate attention computation [47], several specialized kernels have been developed based on FlashAttention [4,5,17,21]. Due to the increasing complexity of modern GPUs—such as tensor cores in A100s [29] and thread block clusters in H100s [7]—manually designed kernels may miss subtle optimizations that are hard to discover manually.

Superoptimization-based approaches. Superoptimization was originally introduced to find optimal instruction sequences [11, 30, 36]. Recent work has applied superoptimization techniques to tensor programs [23–26, 45, 46, 49, 52]. However, all these attempts only consider algebraic transformations at the kernel level and cannot discover more sophisticated optimizations that require jointly considering algebraic and schedule transformations at all of the kernel, block, and thread levels. Our evaluation shows that Mirage largely outperforms existing DNN superoptimizers, demonstrating the importance of multi-level joint optimization.

Schedule-based approaches. Recent work has introduced ML compilers that automatically optimize the execution schedule of kernel GPUs. Systems such as TVM [13, 14], Ansor [51], and Triton [43], along with others [19, 20, 53], build on the idea of algorithm-schedule separation introduced in Halide. They search for optimized schedules to execute a user-specified algorithm on GPUs. However, schedule-based approaches require users to explicitly specify the algorithm for each kernel, and their performance is limited to the quality of these provided algorithms.

Multi-level graph representations. Welder [38] and ASPEN [33] introduce multi-level tile graphs that share similarities with Mirage's μGraphs, as both representations follow the GPU hierarchy. However, prior work focuses on scheduling transformations, while Mirage extends beyond scheduling by also considering algebraic transformations and the discovery of new custom kernels. Most optimizations presented in this paper fall outside the scope of these prior approaches.

10 Conclusion

This paper proposes Mirage, the first multi-level superoptimizer for tensor programs. Mirage introduces a hierarchy graph representation to specify a tensor program at the kernel, thread block, and thread levels of the GPU execution hierarchy, and uses a novel pruning technique based on abstraction to significantly reduce the search space Mirage needs to consider while providing a certain optimality guarantee. Mirage outperforms existing tensor program optimizers by up to 3.3×, even for widely used and heavily optimized DNNs.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Stephanie Wang, for their valuable comments and suggestions. We thank Tianqi Chen, Phillip Gibbons, Bohan Hou, Muyan Hu, Jinchen Jiang, Xiaoyu Jiang, Ruihang Lai, Yu Zhou, and other CMU Catalyst members for their feedback on this work. This research is partially supported by NSF awards CNS-2147909, CNS-2211882, and CNS-2239351, and research awards from Amazon, Cisco, Google, Meta, NVIDIA, Oracle, Qualcomm, and Samsung. This research is also partially supported by a research grant from the Center for New Scientists at the Weizmann Institute of Science and by a grant from the Azrieli Foundation.

References

[1] Xla: Optimizing compiler for tensorflow. https://www.tensorflow.org/xla, 2017. 14

[2] Nvidia/cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2019. 10

[3] Tensorflow graph optimization with grappler. https://www.tensorflow.org/guide/graph_optimization, 2019. 1

[4] Transformer related optimizations. https://github.com/NVIDIA/FasterTransformer, 2020. 14

[5] Flash-decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html, 2023. 2, 14

[6] Llama-7b-lora. https://huggingface.co/Laurie/llama7b-lora-merged/tree/main, 2023. 10

[7] Nvidia h100 tensor core gpu. https://www.nvidia.com/en-us/data-center/h100/, 2023. 14

[8] A Triton implementation of the FlashAttention2 algorithm. https://triton-lang.org/main/getting-started/tutorials/06-fused-attention.html, 2023. 2

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016. 1, 14

[10] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023. 10, 12

[11] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, 2006. 14

[12] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models, 2022. 14

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. 1, 5, 14

[14] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018. 14

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. 3, 10, 11

[16] Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas, 2016. 3, 9, 10, 11

[17] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference, 2023. 2, 11, 12, 14

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008. 7, 9, 10

[19] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. Tensorir: An abstraction for automatic tensorized program optimization, 2022. 14

[20] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 302–313, New York, NY, USA, 2023. Association for Computing Machinery. 14

[21] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus, 2024. 11, 14

[22] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 11

[23] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, and Zhihao Jia. Optimal kernel orchestration for tensor programs with korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 755–769, New York, NY, USA, 2024. Association for Computing Machinery. 14

[24] Byungsoo Jeon, Mengdi Wu, Shiyi Cao, Sunghyun Kim, Sunghyun Park, Neeraj Aggarwal, Colin Unger, Daiyaan Arfeen, Peiyuan Liao, Xupeng Miao, Mohammad Alizadeh, Gregory R. Ganger, Tianqi Chen, and Zhihao Jia. Graphpipe: Improving performance and scalability of dnn training with graph pipeline parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 557–571, New York, NY, USA, 2025. Association for Computing Machinery. 14

[25] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery. 1, 2, 5, 11, 14

[26] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019. 14

[27] Jiatu Li and Mengdi Wu. Identity testing for circuits with exponentiation gates, 2025. 9

[28] Ilya Loshchilov, Cheng-Ping Hsieh, Simeng Sun, and Boris Ginsburg. nGPT: Normalized transformer with representation learning on the hypersphere, 2024. 10, 11, 13

[29] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2018. 14

[30] Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGARCH Computer Architecture News*, volume 15, 1987. 14

[31] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), 2016. 5

[32] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress. 10

[33] Jongseok Park, Kyungmin Bin, Gibum Park, Sangtae Ha, and Kyunghan Lee. Aspen: Breaking operator barriers for efficient parallelization of deep neural networks. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 68625–68638. Curran Associates, Inc., 2023. 14

[34] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017. 1, 11, 14

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013. 1, 5

[36] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGPLAN Notices*, volume 48, 2013. 14

[37] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, oct 1980. 2, 8, 9

[38] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, Boston, MA, July 2023. USENIX Association. 14

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. 11

[40] Chameleon Team. Chameleon: Mixed-modal early-fusion foundation models, 2024. 10, 11, 12

[41] The Llama 3 team. The llama 3 herd of models, 2024. 3, 5, 10, 11

[42] NVIDIA TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017. 11, 14

[43] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery. 2, 11, 14

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023. 10

[45] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022. 14

[46] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021. 1, 2, 5, 8, 11, 14

[47] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax. https://github.com/huggingface/transformers, 2022. 2, 14

[48] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization, 2021. 1

[49] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality Saturation for Tensor Graph Superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, March 2021. 14

[50] Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019. 5

[51] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating high-performance tensor programs for deep learning. *CoRR*, abs/2006.06762, 2020. 1, 14

[52] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, and Zhihao Jia. EINNET: Optimizing tensor programs with Derivation-Based transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 739–755, Boston, MA, July 2023. USENIX Association. 14

[53] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery. 14

[54] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979. 2, 8, 9

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

QiMeng-Xpiler: Transcompiling Tensor Programs for Deep Learning Systems with a Neural-Symbolic Approach

Shouyang Dong[1,2,3], Yuanbo Wen[3], Jun Bi[3], Di Huang[3], Jiaming Guo[3], Jianxing Xu[1,2,3], Ruibai Xu[1,2,3]
Xinkai Song[3], Yifan Hao[3], Ling Li[4,5], Xuehai Zhou[1], Tianshi Chen[2], Qi Guo[3], Yunji Chen[3,5]*

[1]*University of Science and Technology of China,* [2]*Cambricon Technologies*
[3]*SKL of Processors, Institute of Computing Technology, Chinese Academy of Sciences*
[4]*Institute of Software, Chinese Academy of Sciences,* [5]*University of Chinese Academy of Sciences*

Abstract

Heterogeneous deep learning systems (DLS) such as GPUs and ASICs have been widely deployed in industrial data centers, which requires to develop multiple low-level tensor programs for different platforms. An attractive solution to relieve the programming burden is to transcompile the legacy code of one platform to others. However, current transcompilation techniques struggle with either tremendous manual efforts or functional incorrectness, rendering "Write Once, Run Anywhere" of tensor programs an open question.

We propose a novel transcompiler, i.e., *QiMeng-Xpiler*, for automatically translating tensor programs across DLS via both large language models (LLMs) and symbolic program synthesis, i.e., neural-symbolic synthesis. The key insight is *leveraging the powerful code generation ability of LLM to make costly search-based symbolic synthesis computationally tractable*. Concretely, we propose multiple LLM-assisted compilation passes via pre-defined meta-prompts for program transformation. During each program transformation, efficient symbolic program synthesis is employed to repair incorrect code snippets with a limited scale. To attain high performance, we propose a hierarchical auto-tuning approach to systematically explore both the parameters and sequences of transformation passes. Experiments on 4 DLS with distinct programming interfaces, i.e., Intel DL Boost with VNNI, NVIDIA GPU with CUDA, AMD MI with HIP, and Cambricon MLU with BANG, demonstrate that *QiMeng-Xpiler* correctly translates different tensor programs at the accuracy of 95% on average.

1 Introduction

Due to the ever-increasing demand for computation from neural network workloads, various deep learning systems (DLS), e.g., NVIDIA GPU with Tensor Core [12], Google TPU [32], GraphCore IPU [9], and Cambricon MLU [3], have been deployed in data centers of cloud and internet service companies such as Microsoft [10], Google [6], and Amazon [4]. To fully exploit various DLS, it is required to develop multiple high-performance tensor programs, i.e., low-level implementations of tensor operators in deep learning, for different platforms, which are notoriously challenging because of the complicated architecture and programming models. Ideally, if we only need to write one copy of a program, and can run it on different platforms, that is, "Write Once, Run Anywhere", it is feasible to address the programming problem in data centers with heterogeneous DLS.

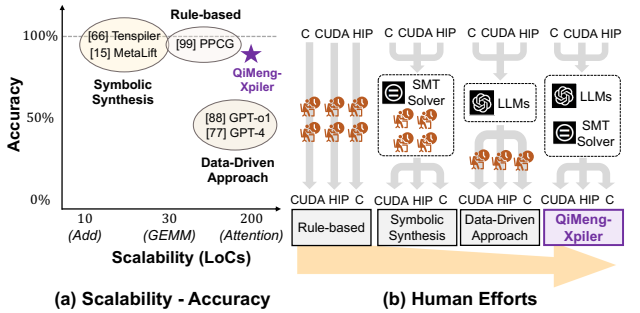


Figure 1: Comparing *QiMeng-Xpiler* to existing transcompilation techniques on (a) scalability-accuracy axis and (b) human efforts

One of the most attractive solutions to achieve "Write Once, Run Anywhere" on heterogeneous platforms is source-to-source compilation, i.e., transcompilation, which automatically compiles the code written with one high-level language to another. Existing transcompilation efforts fall into three categories, i.e., rule-based, symbolic synthesis, and data-driven approaches. The rule-based approaches require experts to manually define a set of transformation rules, typically applied to Abstract Syntax Tree (AST), between different programming languages, and then employ pattern matching to parse the input source programs [26, 31]. For example, a CUDA-to-FPGA translator, i.e., FCUDA, defines a set of transformation rules for data communication, compute optimization, and parallelism mapping [40]. The symbolic synthesis approaches generate semantic-preserving target code from either domain-specific languages [41] or input/output examples [52]. Since they typically rely on a costly search-based SMT (Sat-

*Yunji Chen (cyj@ict.ac.cn) is the corresponding author.

isfiability Modulo Theories) solver [37], it is hard to scale to large-size general-purpose programs. The data-driven approaches, which typically train neural networks from a large amount of source code for code generation, have emerged recently [16,44,53]. Notable examples include TransCoder [43], StarCoder [34], GPT-4 [15], and OpenAI-o1 [8] which automatically translate programs written in different high-level languages such as C++, Java, and Python.

However, the above transcompilation techniques are not applicable to DLS due to their complicated architecture and programming models. Regarding the rule-based approaches, the huge architectural discrepancy between different platforms makes it infeasible to manually define efficient transformation rules. Regarding the symbolic synthesis approaches, in addition to their limited scalability, it is infeasible to handle different parallel semantics, e.g., SIMT in CUDA C and SIMD in BANG C [2], for the SMT solver. Meanwhile, it also requires considerable manual efforts to accurately specify the input constraints for the SMT solver. Regarding the data-driven approaches, even with pre-trained large language models (LLMs), tensor program semantics cannot be fully preserved and thus the translation accuracy only achieves 29.6% [35], which requires considerable manual efforts for code correction. In summary, existing transcompilation techniques for tensor programs face challenges such as tremendous manual efforts, limited scalability, or functional incorrectness, as shown in Figure 1.

To automatically translate tensor programs across DLS with a correctness guarantee, we propose to exploit both the *flexibility of data-driven approaches*, specifically, via LLMs, and the *soundness of symbolic synthesis approaches*. However, the challenge of collaboration between LLM and symbolic synthesis is three-fold: (1) it is difficult to generate highly accurate code via LLM due to the scarcity of tensor programs, (2) it is non-trivial to determine which code snippets should be formally generated by search-based symbolic synthesis, in order to balance the computational costs and achieved accuracy, and (3) performance optimizations are hard to specify either with LLM prompts or synthesis specifications, making the performance of generated code significantly lags behind that of human experts.

1.1 Our Proposal

To address the above challenges, we propose to build a novel transcompiler, i.e., *QiMeng-Xpiler*, for translating tensor programs across DLS via both LLM (i.e., GPT-4 [20]) and search-based symbolic synthesis. The key of *QiMeng-Xpiler* is that *the program translation is automatically conducted as a series of LLM-assisted transformation passes, where incorrect code snippets are repaired by small-scale symbolic synthesis, and the optimal transformation passes are identified via hierarchical auto-tuning*. The advantages of proposed neural-symbolic synthesis are: (1) the entire transcompilation is decomposed into multiple transformation passes via a chain of LLM prompting, instead of a single prompt in traditional LLM-assisted code generation, to significantly improve the accuracy, (2) the problem size for symbolic synthesis is constrained to a limited scale so that the SMT solver can handle it efficiently, and (3) the performance optimization is achieved by exploring both the parameters and sequences of transformation passes.

Concretely, by considering the high parallelism, memory hierarchy, and specialized ISA of DLS, the proposed approach consists of three categories of transformation, i.e., *sequentialization/parallelization*, *memory conversion*, and *(de)tensorization*. Each transformation pass is first performed with LLM-based code transformation, then validated with unit-test, and finally repaired with an SMT-solver if necessary. Then, a hierarchical auto-tuning approach consisting of the *intra-pass* and *inter-pass* auto-tuning is conducted to improve the performance. The intra-pass auto-tuning uses brute-force search to find the optimal parameters (e.g., tiling sizes) for program transformation, and the inter-pass auto-tuning uses MCTS (Monte Carlo Tree Search) [19] to determine the optimal sequence of passes with maximized performance.

To demonstrate the generality of *QiMeng-Xpiler*, we conduct experiments on 4 different DLS with their interfaces, i.e., Intel DL Boost with VNNI intrinsics, NVIDIA GPU with CUDA C, AMD MI with HIP code, and Cambricon MLU with BANG C. Experimental results show that *QiMeng-Xpiler* correctly translates different programs at the accuracy of 95% on average, and the performance of translated programs achieves an average of 0.78× of vendor-provided manually-optimized libraries including cuDNN/cuBLAS [1,11] and oneDNN [14]. Moreover, the programming productivity of 2 representative DLS, i.e., NVIDIA GPU and MLU, is improved by up to 34.3× and 96.0×, respectively.

1.2 Key Contributions

To our best knowledge, this work is *the first to automatically translate tensor programs of deep learning systems with different programming models*. This paper makes the following contributions:

- **Neural-symbolic program synthesis**. We propose to use LLMs for generating high-level program sketches via pre-defined meta-prompts and repair the incorrect low-level details through SMT-based symbolic synthesis with limited scale, which achieves *automatic program translation with a correctness guarantee*.
- **Hierarchical performance auto-tuning**. We propose a hierarchical auto-tuning approach, where the *intra-pass* and *inter-pass* auto-tuning are used for exploring pass parameters and pass sequences, respectively, to maximize the performance of synthesized programs.
- **Extensive evaluation**. We conduct extensive evaluations on 4 different deep learning systems, and experimental

Table 1: Comparison of deep learning systems (Intel DL Boost, NVIDIA GPU, AMD MI, and Cambricon MLU) and their programming interfaces. The characteristics fall into three categories: Parallelism, Memory Hierarchy, and Specialized Intrinsics.

Platforms	Interfaces	Categories	Examples
Intel DL Boost	C with VNNI extensions	Specialized Intrinsic	`_mm_dpbusds_epi32(...)`, `_mm512_dpbusd_epi32(...)`
NVIDIA GPU with Tensor Core	CUDA C	Parallelism	`blockIdx`, `threadIdx`
		Memory Hierarchy	`registers`, `__shared__`, `__global__`
			`matrix_a`, `matrix_b`, and `accumulator` in Tensor Core fragments
		Specialized Intrinsics	`wmma::mma_sync(d, a, b, c)`
AMD MI with Matrix Core	HIP	Parallelism	`blockIdx`, `threadIdx`
		Memory Hierarchy	`registers`, `__shared__`, `__global__`
			`matrix_a`, `matrix_b`, and `accumulator` in Matrix Core fragments
		Specialized Intrinsics	`d = __builtin_amdgcn_mfma_f32_16x16x4f32(a, b, c, ...)`
Cambricon MLU	BANG C	Parallelism	`taskId` for task-level parallelism
			`clusterId`, `coreId` for multi-core parallelism
		Memory Hierarchy	`registers`, `__mlu_shared__`, `__mlu_device__`
			`__nram__`, `__wram__`
		Specialized Intrinsics	`__bang_mlp(...)`, `__bang_conv(...)`

results on accuracy, execution performance, and productivity improvement well demonstrate the effectiveness and efficiency of this work.

2 Background and Motivation

2.1 Background

Programming Deep Learning Systems. Deep learning systems (DLS) adopt different programming languages tailored to their hardware, such as CUDA C for NVIDIA GPU, HIP for AMD MI, BANG C for Cambricon MLU, and VNNI intrinsics for Intel DL Boost CPU. However, programming DLS poses substantial challenges due to their *inherent high parallelism*, *intricate memory hierarchies*, and *specialized instruction sets*. First, DLS typically follows parallel programming models, like SIMT or multi-core, diverging from conventional serial programming. This paradigm shift necessitates a deep understanding of task and data parallelism from programmers. Furthermore, DLS are designed with complex on-chip memory hierarchies to process diverse data types efficiently. For instance, Cambricon MLU uses separate neuron and weight storage for different memory types (i.e., NRAM and WRAM space), while NVIDIA GPU utilizes various memory spaces (e.g., local and shared memory) for multi-level dataflow. Such designs require programmers to explicitly manage complicated memory hierarchies. Additionally, to suit deep learning's computational demands, DLS often incorporate specialized instruction sets. Examples include Intel's DL Boost VNNI for specific operations and Cambricon's tensor intrinsics. These specialized intrinsics often come with intricate constraints, adding extra complexity to programming. Table 1 illustrates the stated programming difficulty for various DLS.

Search-based Program Synthesis. Researchers have studied how to use search-based program synthesis to transform legacy code to high-performance code with minimal manual efforts [17, 18, 30, 49]. The idea is to obtain programs that satisfy semantic correctness by searching through a predefined domain-specific language (DSL) or using an SMT solver for constraint solving. The main advantage of search-based program synthesis is to ensure the semantic equivalence between the original code and the translated code. However, these approaches always struggle with large search spaces, which means that they can only be applied to pre-defined DSL and thus cannot generate code for commodity deep learning systems. Additionally, they can only handle computational instructions and are unable to meet optimization requirements for hierarchical storage and parallel capabilities. Thus, the source-to-source translation across different deep learning systems remains an open problem.

LLM-assisted Program Generation. LLMs such as Codex [21] and StarCoder [34] are being increasingly used to assist in programming. They can generate programs based on problem descriptions and generate possible code snippets based on the provided code context, helping programmers write code quickly. They also allow programmers to describe problems in natural language without following specific programming language syntax. However, these LLMs have several limitations. Firstly, they lack a deep understanding of program semantics and cannot comprehend the intent of the code. This means that the generated code may be incorrect or incomplete, requiring further adjustment and optimization. Secondly, these models perform poorly in specific domains or complex problems, because it is typically difficult to obtain high-quality and diverse training data for these problems.

2.2 Motivation

To understand the design principle of *QiMeng-Xpiler*, we evaluate transcompilation of real-world tensor programs by state-of-the-art LLM (i.e., GPT-4) and program synthesis tools.

Taxonomy of the transcompilation errors. We first classi-

fied all errors introduced in the transcompilation process into 3 categories, each corresponding to specific characteristics of DLS mentioned above: (1) *Parallelism-related errors*, where the transcompilation fails to analyze the loop semantics and generates the incorrect loops or builtin variables of DLS, (2) *Memory-related errors*, where the transcompilation cannot deal with the intricate memory hierarchies for the DLS, resulting in incorrect memory declarations and usage, and (3) *Instruction-related errors*, which occur when the translated code utilizes incorrect instructions or parameters that cannot perform the same computation. Figure 2 illustrates examples of different errors when transcompiling from CUDA C to BANG C code by GPT-4. In Figure 2(a), LLM blindly reuses the GPU's parallel index (e.g., blockIdx.x * 1024 + threadIdx.x) on the MLU, ignoring the fact that the MLU's clusterId and coreId dimensions differ. Figure 2(b) fails to correctly place the B Tensor in the original GEMM operation into the required memory hierarchy, as `__bang_mlp` expects the B to be stored in WRAM, not NRAM. Figure 2(c) tends to replace the original SIMT-based scalar operations with SIMD-based tensorized instructions, but the parameter that indicates the tensor length should be 2309 rather than 1024.

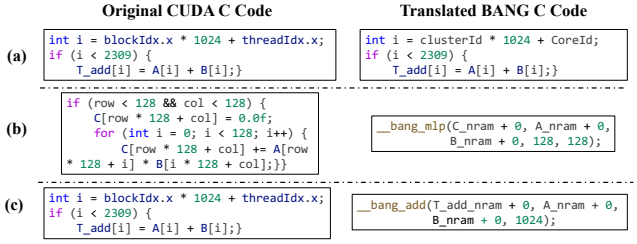


Figure 2: The unsuccessful transcompilation examples of GPT-4: (a) Parallelism-related. (b) Memory-related. (c) Instruction-related.

Based on the transcompilation error taxonomy, we examine the existing methods and draw the following observations.

Observation #1: *The single-step LLM-based transcompilation exhibits significant error rates across all three categories, posing challenges for accurately transcompiling programs across DLS.*

Table 2 presents the detailed results for transcompiling CUDA C code to BANG C for real-world operators (detailed in Sec.7) using GPT-4. Note that these categories are not mutually exclusive, which means a single program may trigger multiple types of errors. For the zero-shot, the compilation error rate was 100%, primarily because the LLM struggled with the *complex memory hierarchy* (100%) and *special instructions* (100%) due to its insufficient training on DLS-specific datasets. For the few-shot which provided several examples in prompts, although the translated code compiled successfully, it showed functional errors, particularly with parallelism (97.2%) and instruction-related issues (94.4%), resulting in an overall computation error rate of 92.3%. These findings highlight the limitations of single-step LLM-based transcompilations, even with the most advanced LLMs.

Observation #2: *Search-based program synthesis excels at optimizing loop bounds and indexing but struggles with high-level program sketches generation, while LLMs are proficient at generating high-level program sketches but prone to errors in low-level details such as loop bounds and indexing.*

Table 2: Breakdown of the unsuccessful transcompilations produced by GPT-4 based on outcome.(%)

Errors	Compilation			Total	Computation			Total
	Parallelism	Memory	Instruction		Parallelism	Memory	Instruction	
Zero-Shot	3	100	100	100	–	–	–	–
Few-Shot	2.3	27.1	76.5	49.4	97.2	2.8	94.4	92.3

Table 3: Taxonomy of the search-based program synthesis approach.

	Method	Inputs	Solving Time
High-level Program Sketchs	Verified Lifting (Tenspiler [42])	Custom IR	+++
Low-level Program Details	SMT Solver (Z3 [37])	SMT Query	+

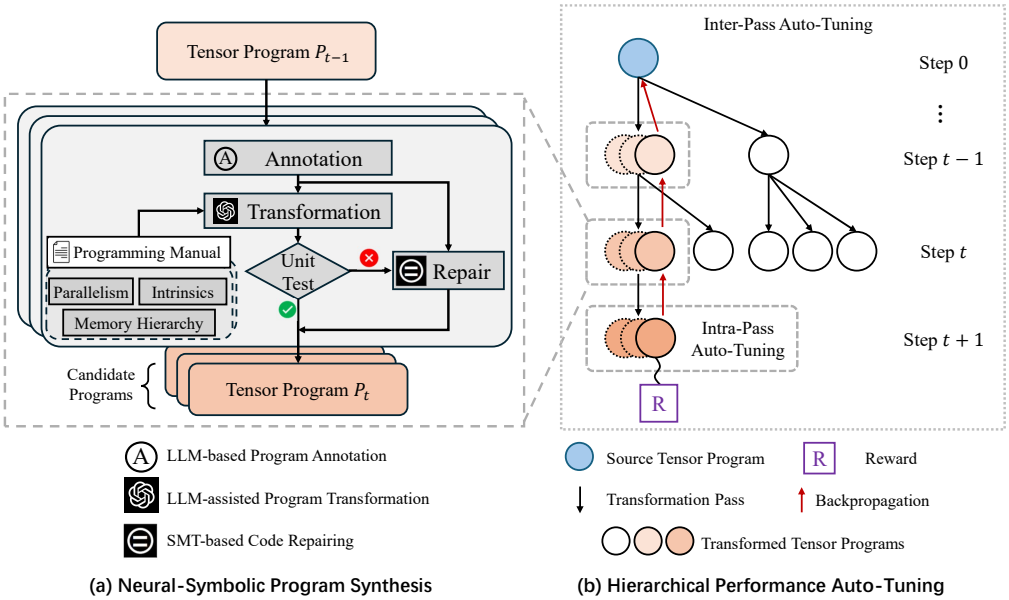
We analyze the strengths and limitations of search-based program synthesis and LLMs in tensor program transcompilation, highlighting their complementary roles. Search-based synthesis, as shown in Table 3, excels at synthesizing and verifying low-level details such as loop bounds and indexing, leveraging its ability to handle mathematical constraints. However, it struggles with generating high-level structures like control flow and efficiently utilizing the memory hierarchy and specialized intrinsics. In contrast, LLMs are highly effective at generating high-level program skeletons but prone to errors in low-level details, such as loop bounds and indexing. As illustrated in Figure 2(c), although the LLM correctly captures the semantics and utilizes the `__bang_add` intrinsic, it mistakenly sets the tensor length to 1024 instead of passing the actual scalar loop bound (i.e., 2309). By combining both approaches, LLMs generate the high-level program structure, while search-based synthesis refines the low-level details, leading to more accurate, efficient, and reliable program synthesis.

These observations motivate us to decompose the program translation process into a series of LLM-assisted transformations, where program synthesis tools can further enhance the correctness of transcompilation.

3 Overview

To automatically translate tensor programs across DLS with a correctness guarantee, as shown in Figure 3, *QiMeng-Xpiler* consists of two parts, i.e., *Neural-Symbolic Program Synthesis* and *Hierarchical Performance Auto-Tuning*.

Neural-Symbolic Program Synthesis. *QiMeng-Xpiler* first decomposes the entire transcompilation process into a series of LLM-assisted transformation passes. Among these, the 11 passes as listed in Table 4 can be categorized into 3 classes, i.e., (1)*sequentialization/parallelization*, (2)*memory conversion*, and (3)*(de)tensorization*. Specifically, the sequentialization/parallelization passes directly convert a parallel program into its sequential counterpart (e.g., from CUDA C to C) or vice versa, by appropriately mapping built-in parallel variables (e.g., `threadIdx.x` in CUDA C) to indexing vari-



(a) Neural-Symbolic Program Synthesis (b) Hierarchical Performance Auto-Tuning

Figure 3: The overview of *QiMeng-Xpiler*, a novel transcompiler for automatic transcompilation of tensor programs across different programming models. The transcompiler consists of two parts: (a) neural-symbolic program synthesis, which utilizes LLM to transform code and repair incorrect transformation through symbolic synthesis with limited scales, and (b) hierarchical performance auto-tuning, which systemically explores both the parameters and sequences of transformation passes.

Table 4: The transformation passes and their description.

	Name	Decription
(1)	Loop Recovery	Convert parallel variables to sequential for loops
	Loop Bind	Assign a sequential loop to parallel variables
	Loop Split	Divide a loop into several sub-loops
	Loop Fuse	Merge several loops into a hyper-loop
	Loop Reorder	Change the execution orders of loops
	Loop Expansion	Split a loop body into several loop bodies
	Loop Contraction	Merge the producer in the loop body of consumer
(2)	Cache	Adapt to the memory hierarchy for efficient load/store inputs/outputs
	Pipeline	Pipeline of data load/store and computation
(3)	Tensorize	Replace a specific loop body to leverage special intrinsics
	Detensorize	Restore a specific loop body from special intrinsics

ables. The memory conversion passes bridge the semantic gap between the memory hierarchies of different DLS by employing suitable strategies for data movement and access patterns. The (de)tensorization passes either convert sequential code into its parallel equivalent by invoking corresponding tensor intrinsics, automatically retrieved from the target system's programming manuals, or restore the original sequential code from the tensor intrinsics. Overall, these passes, each addressing a key characteristic of different DLS, i.e., parallelism, memory hierarchy, and specialized intrinsics, are sufficient to perform tensor program transcompilation across DLS.

To ensure both flexibility and correctness, each program transformation pass in *QiMeng-Xpiler* adopts a neural-symbolic approach, i.e., utilizing LLM to transform code and repair incorrect transformation through symbolic synthesis with limited scales. In each transformation pass, the LLM first generates transformed code, and this transformed code is then validated by unit tests; if it fails, it is repaired using small-scale SMT-based symbolic synthesis. We will detail this in Section 4.

Hierarchical Performance Auto-Tuning. To achieve automatic transcompilation and maximize the performance of transformed programs, *QiMeng-Xpiler* employs a hierarchical auto-tuning approach to systemically explore both the parameters (intra-pass auto-tuning) and sequences (inter-pass auto-tuning) of transformation passes. Specifically, the intra-pass auto-tuning utilizes brute-force search to identify the optimal parameters (e.g., tiling sizes) for program transformation. The inter-pass auto-tuning, on the other hand, leverages MCTS to automatically discover both functionally correct and optimal sequences of transformation passes for translating tensor programs across different DLS. Concretely, the MCTS in the inter-pass auto-tuning consists of four components: tensor programs, transformation pass, backpropagation, and reward. At each step $t-1$, the MCTS selects a candidate node (i.e., tensor program P_{t-1}) based on its score and expands the node by transforming it with an available transformation pass, resulting in tensor program P_t. Then, by executing P_t, a reward related to execution time is obtained. This reward is backpropagated to all ancestor nodes along the expansion path, updating their scores for the selection in the next step t. We will detail this in Section 5.

4 Neural-Symbolic Program Synthesis

QiMeng-Xpiler breaks down the traditional program transformation process into multiple transformation passes, with each pass covering a full workflow of neural-symbolic program synthesis containing program annotation, meta-prompts based transformation, bug localization, and SMT-based code repairing. The neural processes, program annotation, and meta-prompts based transformation, leverage the program-

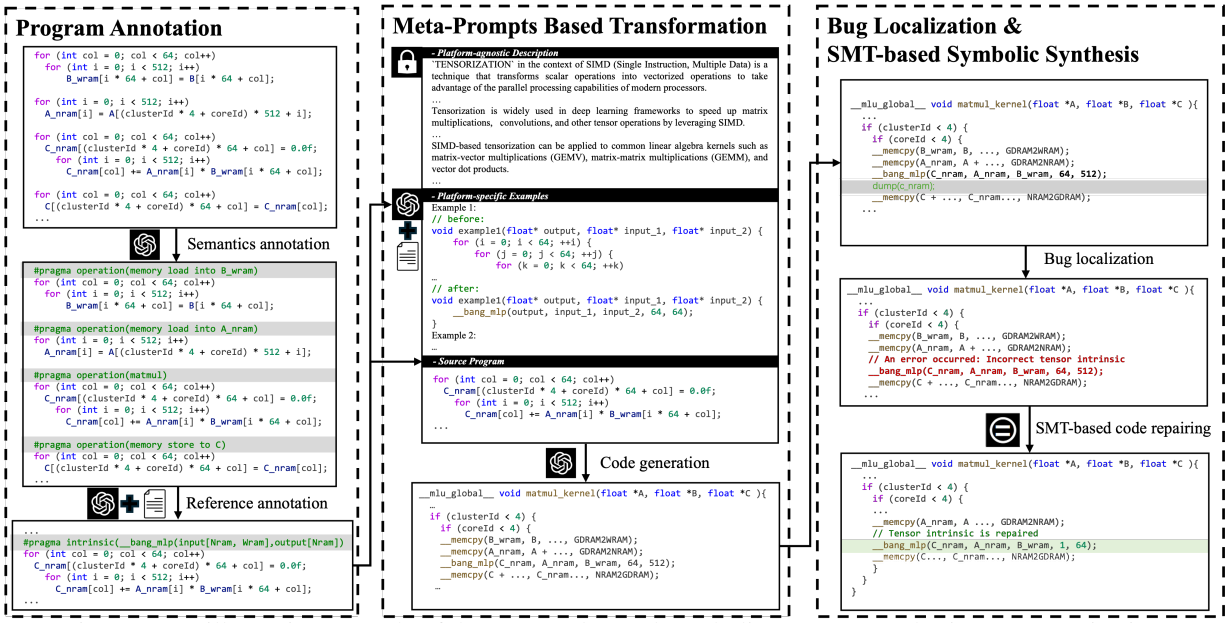


Figure 4: An illustrative example of the proposed neural-symbolic program synthesis on a tensorization case.

ming manual and the prior knowledge within the LLM to enhance the flexibility of program transformation, enabling *QiMeng-Xpiler* to handle larger-scale code compared to traditional methods. Meanwhile, the symbolic processes, bug localization, and SMT-based code repairing, ensure the correctness of the transformed program. These four processes are illustrated in Figure 4.

4.1 Program Annotation

For a specific transformation pass, the program annotation process marks code blocks with semantics related to the transformation pass, assisting in the subsequent program transformation process.

As shown in Algorithm 1, program annotation consists of two parts. The first part (lines 2-3) is semantics annotation which involves identifying computational operations with semantics (e.g., `operation(matmul)`). This part is performed by an LLM. The second part (lines 4-11) is reference annotation which involves performing an information retrieval in the programming manual based on each identified computational operation to obtain the corresponding target operation and parameters (e.g., `intrinsic(__bang_mlp(input[Nram, Wram], output[Nram])`). This part is first performed by a BM25 search engine [48] which can retrieve related information from the programming manual. Then, the retrieved information and the source program are sent to an LLM to annotate the operator with references.

Program annotation serves two main purposes. First, by identifying the computation operations in the source program, semantics annotation abstracts the platform-agnostic functional semantics of the source program instead of focusing on platform-specific details, which can help LLMs generate semantically correct programs in the subsequent LLM-assisted program transformation. Second, by applying reference annotation, the identified computational operations and their associated hardware characteristics (e.g., memory or computation shape constraints for special intrinsics) can be used in the subsequent meta-prompts based transformation process, aiding LLMs in generating more accurate programs.

Algorithm 1: Program Annotation Algorithm

```
   Input  : A source program P
   Output : An annotated program
1  Given: LLM, BM25 search engine (SE), programming manual M
   //Annotate program with identified computation
2  P_d ← LLM(P)
   //Get the list of computation information
3  L ← TraverseComputation(P_d)
4  for n in L do
       //Retrieve the programming manual of each
         computation
5      D ← SE(n, M)
       //Annotate program with memory spaces or tensor
         intrinsics
6      P_d ← LLM(P_d, D)
7  end
8  return P_d
```

4.2 Meta-Prompts based Transformation

Meta-prompts-based transformation uses LLMs to convert the source program into the target program with a correct program sketch. In this approach, meta-prompt is a high-level prompt template that can be adapted to different source programs based on the results of the previous program annotation process. The adapted prompt, along with the source program, is then provided as input to the LLMs. Each transformation pass has its own meta-prompt. Meta-prompts mainly consist of three parts: platform-agnostic description, platform-specific examples, and tuning knobs.

Platform-agnostic description. The platform-agnostic part describes the program's functionality and detailed constraints that must be considered during implementation. For example, in the case of Figure 4, the platform-agnostic part

refers to the functionality of the tensorization in the context of SIMD and the application scenarios like deep learning frameworks and common linear algebra kernels. This part of the prompt remains the same across different platforms.

Platform-specific examples. Although operators on different platforms share similar functional semantics, their implementation details, e.g., function names and parameters, can vary significantly. As a result, directly transforming programs across platforms with LLMs often leads to incorrect programs. To address this, we utilize the annotated source program and the LLM to search for platform-specific implementation examples in the target platform's programming manual that are functionally related to the source program. These examples are incorporated into the meta-prompt as the platform-specific component, bridging the gap between the high-level functional semantics of the operators and target platform. For instance, in Figure 4, the search results include an example of how to transcompile a `matmul` from C to BANG C.

Tuning knobs. The loop split pass and loop reorder pass require more constraints related to the target platform. This information is included in the meta-prompt optionally. For example, loop split needs to determine the loop split alignment size for the target platform and use this as the minimum unit to expand into a corresponding search space, which is then used for the auto-tuning of the generated program. Please refer to details in Section 5.

Based on the meta-prompt, the LLM can generate the transformed code with a generally correct program sketch but error-prone values. Then, we fix these buggy values with symbolic approaches, including bug localization and SMT-based code repairing, which will be introduced as follows.

4.3 Bug Localization

The feasibility of *QiMeng-Xpiler* relies on effectively reducing the code repair problem scale through precise bug localization during unit testing. The process begins by validating the transformed code against the provided unit tests. When a test failure occurs, Algorithm 2 is employed to systematically identify faulty code segments. The localization mechanism first traverses all buffers and applies binary search to pinpoint the initial faulty buffer. For each candidate buffer during the search, the algorithm identifies the corresponding pre-transformation buffer through name similarity matching, then compares the buffer values to determine the search direction. Value matches indicate downstream errors, prompting forward search, while mismatches suggest upstream errors, triggering backward tracing. This iterative process continues until the exact faulty buffer is identified, which subsequently helps determine the buggy code block. For the identified buggy code block, we perform further analysis by categorizing errors into two distinct types: index-related errors (such as inconsistent loop bounds, out-of-bounds accesses, or misaligned indices) and tensor instruction-related errors (including incorrect tensor intrinsic parameters or invalid instructions). Index-related errors are detected through control-flow graph (CFG) analysis, while tensor instruction-related errors are inferred when the CFG appears correct but tensor instructions are present. This localization approach is feasible because each transformation pass introduces only small-step changes while largely preserving the control flow and buffer structure, enabling effective comparative analysis as described.

Figure 4 demonstrates a bug localization example, where inserting a dump function after computing `C_nram` reveals a mismatch, while the input, i.e., `A_nram` and `B_wram` remain correct. This discrepancy confirms that the bug resides in the computation of `C_nram`. Since no control-flow changes are involved, the error is precisely attributed to the `bang_mlp` instruction.

Algorithm 2: Bug Localization Algorithm

Input : Source program S, Transcompiled program E
Output : Faulty code blocks N_e with error type

$S_{\text{ast}} \leftarrow \text{BuildAST}(S)$; $E_{\text{ast}} \leftarrow \text{BuildAST}(E)$
`//Step 1: Faulty Buffer Localization`
$B_S \leftarrow \text{ExtractBufferSequence}(S_{\text{ast}})$;$B_E \leftarrow$ $\text{ExtractBufferSequence}(E_{\text{ast}})$
$b_{\text{err}} \leftarrow \emptyset$
for $b_e \in \mathit{BinarySearch}(B_E)$ **do**
  $b_s \leftarrow \text{MatchByNameSimilarity}(b_e, B_S)$
  if $\mathit{CompareBufferValues}(b_e, b_s) = \mathit{False}$ **then**
    $b_{\text{err}} \leftarrow b_e$
    break
  end
end
`//Step 2: Code Block Mapping`
$N_E \leftarrow \text{FindBufferAccessNodes}(E_{\text{ast}}, b_{\text{err}})$
$N_S \leftarrow \text{MatchControlFlowBlocks}(S_{\text{ast}}, N_E)$
`//Step 3: Error Type Classification`
foreach $(N_s, N_E) \in N_m$ **do**
  if $\mathit{CompareCFG}(N_S, N_E) \neq \emptyset$ **then**
    $T_{\text{err}} \leftarrow \text{IndexError}$
  else if $\mathit{HasTensorIntrinsic}(N_e)$ **then**
    $T_{\text{err}} \leftarrow \text{TensorInstructionError}$
  end
  return (N_e, T_{err})
end

4.4 SMT-based Code Repairing

Once the bug is located, *QiMeng-Xpiler* introduces SMT-based methods for code repairing. For index-related errors, *QiMeng-Xpiler* leverages an SMT solver to perform automatic repairs. As illustrated in Algorithm 3, *QiMeng-Xpiler* constructs symbolic constraints over loop bounds and buffer access indices, and encodes them as SMT queries. The SMT solver then searches for solutions that satisfy these constraints. Figure 5 presents examples of such constraints in two representative scenarios: in loop splitting, the solver guarantees that the transformed loops cover the same iteration space; in cache read insertion, it must align reused access indices. It is worth noting that when the loop nest is deep or the control flow is highly complex, the generated constraints can become too large to solve efficiently. For tensor instruction-related errors, *QiMeng-Xpiler* extracts the scalar computation logic and its corresponding semantics and passes them to Tenspiler [42], which automatically synthesizes semantically equivalent target code containing the appropriate tensor instruction.

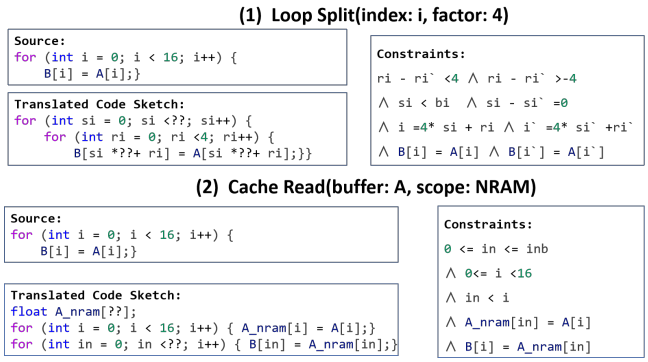


Figure 5: The SMT constraints for loop split and cache read.

By applying the above repair strategies, *QiMeng-Xpiler* effectively corrects LLM-generated programs and outputs the fixed version as the final result of a transformation pass.

Algorithm 3: SMT-based Code Repairing Algorithm

Input : A source program P, error program E
Output : A repaired program F
Given: Error code snippet S
$T_s \leftarrow \text{AST}(P)$; $S_s \leftarrow \text{extract}(T_s, S)$
$T_e \leftarrow \text{AST}(E)$; $S_e \leftarrow \text{extract}(T_e, S)$
//Generate the sketch according to transformation definition
$S_k \leftarrow \text{GenerateCodeSketch}(S_s, S_e)$
//Generate the smt query
$Q \leftarrow \text{CreateSMTQuery}(S_s, S_e)$
//Generate the verified code snippet
$R \leftarrow \text{SynthesisCode}(Q)$
$F \leftarrow \text{StitchBack}(P, R, S)$
return F

5 Hierarchical Performance Auto-Tuning

To fully automate code transcompilation and achieve higher performance, *QiMeng-Xpiler* employs a hierarchical auto-tuning approach, which effectively decouples high-level code structures from low-level details, enabling flexible enumeration of high-level program sketches (i.e., strategizing parallelization and thread bindings), as well as efficient sampling of low-level specifics (i.e., tiling size and loop orders). As a result, transcompilation exploration space can be formalized as follows:

$$S = \left\{ S^{(n)} \mid \begin{array}{l} S^{(t)} = \text{apply}(\text{apply}(S^{(t-1)}, d_t), k_t), \\ \forall d_t \in D_t, 1 \le i \le n, \\ \forall k_t \in K_t, 1 \le t \le n \end{array} \right\} \quad (1)$$

where s^0 denotes the source tensor program, and d_t represents a random selected pass from the pre-defined passes D_t, k_t represents a tuning option from the pre-defined tuning knobs K_t, and n is number of transformation. Thus, the size of the exploration space aligns with the number of pass sequences and tuning knobs. We have:

$$|S| = |D_1| \times |K_1| \times |D_2| \times |K_2| \times \ldots \times |D_n| \times |K_n| \quad (2)$$

To explore this huge search space, *QiMeng-Xpiler* puts forward a hierarchical performance auto-tuning consisting of two main components creatively: (1) **Intra-Pass Auto-Tuning**, which focuses on fine-tuning the performance of translated programs; and (2) **Inter-Pass Auto-Tuning with MCTS**, which is responsible for constructing a vast valid search space to identify the most optimal code transcompilation.

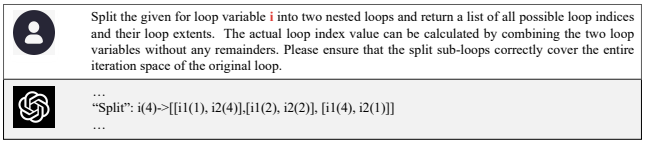


Figure 6: The auto-tuning prompt for loop split.

5.1 Intra-Pass Auto-Tuning

During the transformation process, passes such as `loop split` and `loop reorder` are tasked with making a series of critical decisions, including the split size and loop order, which significantly affect the performance of the resulting tensor program. To address these challenges, we consider implementing auto-tuning, an effective technique for automatically generating high-performance programs. Specifically, auto-tuning is implemented as a functional module that interacts with the aforementioned passes to create a search space consisting of multiple candidate programs. It then explores this space to identify the optimal program configuration.

Search space generation. The basis of intra-pass auto-tuning is to construct a search space with a large number of candidate programs, and our approach generates such space with the help of LLM and programming manuals. As shown in Figure 3, the auto-tuning module first interacts with the *loop split* to generate multiple programs with different adjustable parameters including: (1) the number of blocks for each logical loop, (2) the loop order after making decisions on (1). This is achieved by using specially designed meta-prompts as shown in Figure 6, which are then transferred to the prompts of the loop split and loop reorder pass. Then, the intra-pass auto-tuning module interacts with other passes to generate different program sketches, such as different loop binding strategies.

Search space exploration. Depending on the characteristics of the target code, e.g., in the case of BANG C language with larger instruction granularity resulting in a small search space, we directly employ a brute-force search approach for exploring the program with optimal performance. Actually, the size of the search space varies significantly for different DLS. Take the Matmul operation ($512 \times 512 \times 512$) as an example, on GPU the size of search space (K in Equation 1) is 150 while on MLU the size is only 10.

5.2 Inter-Pass Auto-Tuning with MCTS

We employs MCTS, which formulates the transcompilation as a Markov decision process where each intermediate program is represented as a state and the actions are the different passes that could be applied next. The reward is proportional to the execution time improvement, but any transformation that fails a unit test yields a reward of zero. The solution would be the actions that lead to the optimal pass sequence.

Reward function. To enhance the accuracy of the cost

model while minimizing additional overhead, we add real execution time measurements each time a new transformation pass is declared. While the program is not fully translated, MCTS runs in parallel with the current programs, records the best real throughput as the reward of the current transformation sequence and its corresponding program. At step t, the reward function is

$$T_{ti} = \begin{cases} T(p_{ti}), & \text{if } p_{ti} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

$$R_t = \max(T_{ti}) \quad (4)$$

where p_t denotes the transformed programs at iteration t, i is the index of transformed program within intra-pass auto-tuning search space, and T is the throughput of program.

Searching parameters setting. MCTS may fail to effectively reach a terminal node in a given iteration or even fail to generate the final transcompiled program. Therefore, to determine the appropriate maximum search depth(i.e., n in Equation 1) and the total number of simulations, we conducted design space exploration to balance search time and reward. Specifically, the maximum search depth of MCTS should exceed 11 passes, as all passes are crucial and may be repeated, with the number of simulations directly impacting both search time and reward. Ultimately, we selected $N = 13$ and 512 simulations for MCTS with early stopping mechanism, achieving competitive performance of the transcompiled program within a reasonable search time (e.g., hours).

6 Implementations

6.1 Implementation Details

QiMeng-Xpiler is implemented with $\sim 35k$ lines of Python code, encompassing the following core modules: LLM-based program annotation, LLM-assisted program transformation passes, an automated compilation, verification, and performance-benchmarking framework for bug localization, SMT-based code repair, an MCST search algorithm for inter-pass auto-tuning, and a search module for intra-pass auto-tuning. In addition, we constructed a test suite of $\sim 38k$ lines of CUDA C, HIP, C with VNNI, and BANG C kernel code, with 85.7% of the test cases sourced from TVM-generated kernels and the rest (e.g., *LLM* operation `Deformable Attention`, `Self Attention` and `RMSNorm` as listed in Table 6) sourced from open-source GitHub repositories, to comprehensively evaluate *QiMeng-Xpiler*'s transcompilation performance across multiple DLS.

6.2 Expanding to New DLS

QiMeng-Xpiler consists of two parts: *Neural-Symbolic Program Synthesis* and *Hierarchical Performance Auto-Tuning*. The latter maintains platform independence, enabling seamless expansion to new DLS without modification. The former currently implements 11 transformation passes, each pass consists of up to four processes: **Program Annotation**, **Meta-Prompts based Transformation**, **Bug Localization**, and **SMT-based Code Repairing**, as described in Section 4.

Table 5: Manual effort required for *QiMeng-Xpiler*.

Pass	Processes			
	Annotation	**Transformation**	**Bug Localization**	**Repair(SMT)**
Loop Recovery Loop Bind	–	Specify threads or cores if needed	Auto	Specify threads or cores if needed
Loop Split	–	Auto	Auto	Auto
Loop Fuse	–	Auto		
Loop Reorder	–	Auto		
Loop Expansion	–	Auto		
Loop Contraction	–	Auto		
Pipeline	–	Provide examples if needed		
Detensorize	–	Provide examples if needed		
Cache	Auto	Specify memory space if needed		
Tensorize	Auto	Provide examples if needed	Auto	Extend Tenspiler for new DLS

We report the manual effort required for each process when adapting *QiMeng-Xpiler* to a new DLS, shown in Table 5. In the table, entries marked with "Auto" indicate that the process is fully automated without requiring manual intervention, while "–" means the process does not apply to that particular pass. Concretely, the passes also fall into two categories: platform-agnostic passes and platform-specific passes. Platform-agnostic passes (i.e., Loop Split, Loop Fuse, Loop Reorder, Loop Expansion, and Loop Contraction) primarily operate on loop structures, do not depend on platform semantics, and can be applied across all target platforms using the same prompts and universal examples. Platform-specific passes like Loop Recovery, Loop Bind, Pipeline, Tensorize, Detensorize, and Cache, rely on the hardware characteristics of the target platform. Specifically, in cases where *QiMeng-Xpiler* cannot automatically reliably extract parameters from the DLS manual (i.e., parallel variables for Loop Recovery and Loop Bind, or the memory scope (e.g., local buffer) used for reads and writes in the Cache pass), users need to explicitly specify these details. For example, we manually specify the number of clusters and cores for MLU and define memory scopes across all platforms. For more complex transformations like Pipeline, Tensorize, and Detensorize, representative code examples are essential for improving transformation quality. While *QiMeng-Xpiler* attempts to automatically extract such examples from programming manuals, manual efforts become necessary when the available examples are insufficient or ineffective. For example, we provide Matrix Core samples for Tensorize on AMD MI. Furthermore, during *Tensorize* passes, LLMs may generate incorrect tensor instructions or parameters. We address this by employing Tenspiler [42] for automated tensorization repairing, thus its code generation backend requires extension when adapting to new DLS targets.

Note that this porting effort is only a one-time task with minimal overhead. For example, specifying the parallel vari-

ables and memory scope typically demands just a single additional line prompt. Similarly, extending Tenspiler for instructions like TensorCore, MatrixCore, or AVX-VNNI typically requires only a few lines of code. Once this setup is complete, *QiMeng-Xpiler* can fully support the new platform through its end-to-end fully automated transcompilation flow without requiring further user intervention.

7 Evaluation Methodology

Evaluated Platforms. We conduct experiments on 4 different DLS with distinct programming models as follows: 1) **Intel Gold 6348 CPU with VNNI Extension**, which utilizes a special instruction set for deep learning; 2) **NVIDIA A100 GPU with CUDA C**, which follows the SIMT programming model; 3) **AMD MI200 with HIP**, which provides an alternative to CUDA language; and 4) **Cambricon MLU with BANG C**, which follows SIMD programming model on a DSA.

Evaluated Benchmarks. We evaluate *QiMeng-Xpiler* on 21 widely-used deep learning operators (shown in Table 6), which can be grouped into 6 types of operations including *MatMul*, *Convolution*, *Activation*, *Pooling*, *Element-wise*, and *LLM* operation. Each operator is further evaluated by 8 typical shapes extracted from real network networks such as GPT [39], LLaMA-2 [47], DAT [54], BERT [25], ResNet [27], and MobileNet [28], VGG [45] and thus there are 168 test cases in total for evaluation. The test cases range from 7 to 214 lines of code and cover various types and shapes from real-world applications, which can comprehensively assess *QiMeng-Xpiler*'s capabilities.

Table 6: Evaluated Benchmark for *QiMeng-Xpiler*

Type	Operators	Lines of Code				Cases
		CUDA C	BANG C	Hip	C with VNNI	
MatMul	GEMM, GEMV Batch GEMM	26, 12 29	15, 11 16	25, 12 28	41, 34 47	24
Convolution	Conv1D Conv2D NHWC, NCHW Depthwise Conv	10 32, 30 27	14 23, 30 24	10 32, 30 27	9 83, 85 17	32
Activation	ReLU, Softmax GeLU, Sigmoid	7, 24 7, 10	18, 18 29, 30	7, 24 7, 10	14, 28 14, 14	32
Elementwise	Add, Sign	20, 12	26, 29	18, 12	17, 14	16
Pooling	MaxPool, AvgPool MinPool, SumPool	25, 30 23, 29	25, 25 25, 25	25, 30 23, 29	31, 34 33, 30	32
LLM	LayerNorm Deformable Attention Self Attention, RMSNorm	42 139 54, 25	35 191 64, 18	42 139 54, 25	46 214 111, 18	32

Comparison Baselines. The comparison baselines for evaluations on accuracy include state-of-the-art LLM-based approaches and rule-based approaches. Regarding the LLM-based approaches, we select the **GPT-4 Zero-Shot**, **GPT-4 Few-Shot**, **OpenAI o1 Zero-Shot**, and **OpenAI o1 Few-Shot** as baselines for each transcompilation direction. Regarding the rule-based approaches, we compare *QiMeng-Xpiler* with two established approaches: **PPCG** [50] and **HIPIFY** [7]. PPCG utilizes polyhedral models for auto-parallelization from C to CUDA HIPIFY is a vendor-provided tool to migrate CUDA code to AMD HIP code. Note that rule-based approaches are not available for other transcompilation directions, primarily due to the complexity and significant human effort required in developing such tools. In contrast, search-based program synthesis methods struggle with the large search space and therefore are not suitable for direct transcompilation across different DLS. Furthermore, we evaluate execution performance by comparing the code generated by *QiMeng-Xpiler* with its manually optimized counterparts (e.g., PyTorch with backend libraries such as cuDNN/cuBLAS, CNNL, rocBLAS, and oneDNN). All experiments were conducted using PyTorch as the baseline framework. The corresponding backend libraries of Pytorch for each platform are listed in Table 7.

Table 7: The versions of the performance comparison baselines.

Baselines	Version	Description
PyTorch	v2.5	Deep Learning framework
cuDNN	v9.2.1	Backend library for NVIDIA GPU
CNNL	v2.0.3	Backend library for Cambricon MLU
OneDNN	v3.6	Backend library for Intel DL Boost
rocBLAS	v4.4.0	Backend library for AMD MI

Evaluation Metrics. We use three key metrics to evaluate the performance and accuracy of *QiMeng-Xpiler*: (1) **Compilation accuracy** is the ratio of programs that are correctly compiled. This metric gauges *QiMeng-Xpiler*'s ability to successfully pass compiler checks, reflecting its proficiency in processing source code to generate error-free programs. (2) **Computation accuracy** is introduced in this paper as a crucial metric that assesses the functional correctness of the translated code, deeming a generated code correct if it passes a set of unit tests. (3) **Execution performance** evaluates the performance of the translated code, highlighting the significance and practicality of the proposed method in real-world applications. In these metrics, we execute the translated code on real platforms and verify its correctness.

8 Experimental Results

8.1 Evaluations on Accuracy

We present the evaluations on compilation/computation accuracy in Table 8 and Table 9, where we compare *QiMeng-Xpiler* with state-of-the-art methods in different transcompilation directions. We conclude that **(1) *QiMeng-Xpiler* performs the best in all directions with close to 100% accuracy for compilation and 86.9% to 100% accuracy for computation.** This clearly indicates that *QiMeng-Xpiler* is capable of handling source-to-source code translation tasks on various DLS with minimal human efforts, bringing revolutionary advancements to the DLS programming domain. **(2) *QiMeng-Xpiler* performs better than the SOTA LLMs** (Table 8). While the SOTA LLMs have achieved high accuracy in some cases, their inherent uncertainty makes it impossible to guarantee 100% correctness. This limitation prevents them

Table 8: Experimental results on different transcompilation directions. (%)

Source	Method	Compilation Accuracy				Computation Accuracy			
		CUDA C	BANG C	Hip	C with VNNI	CUDA C	BANG C	Hip	C with VNNI
CUDA C	GPT-4 Zero-Shot	–	0	82.7	9.5	–	0	82.7	4.2
	OpenAI o1 Zero-Shot	–	0	85.7	61.9	–	0	82.7	60.7
	GPT-4 Few-Shot	–	50.6	97.0	84.5	–	7.7	96.4	30.4
	OpenAI o1 Few-Shot	–	51.8	98.2	85.1	–	48.2	98.2	55.4
	QiMeng-Xpiler w/o SMT	–	82.7	98.2	88.1	–	54.2	98.2	58.3
	QiMeng-Xpiler w/o SMT + Self-Debugging	–	87.5	98.8	89.3	–	54.8	98.2	58.9
	QiMeng-Xpiler	–	**100**	**100**	**100**	–	**91.7**	**100**	**95.2**
BANG C	GPT-4 Zero-Shot	24.4	–	26.8	0	0	–	0	0
	OpenAI o1 Zero-Shot	27.4	–	97.0	9.5	0	–	0	4.2
	GPT-4 Few-Shot	69.0	–	66.1	23.8	6.5	–	6.5	13.1
	OpenAI o1 Few-Shot	71.4	–	97.0	41.7	10.1	–	7.7	23.2
	QiMeng-Xpiler w/o SMT	85.1	–	84.5	47.6	77.4	–	78.6	41.1
	QiMeng-Xpiler w/o SMT + Self-Debugging	88.1	–	88.7	50.6	77.4	–	78.6	41.1
	QiMeng-Xpiler	**100**	–	**100**	**100**	**95.8**	–	**97.0**	**95.2**
Hip	GPT-4 Zero-Shot	97.0	0	–	23.8	97.0	0	–	5.4
	OpenAI o1 Zero-Shot	98.2	0	–	45.8	98.2	0	–	4.2
	GPT-4 Few-Shot	97.0	35.1	–	85.1	97.0	5.4	–	24.4
	OpenAI o1 Few-Shot	98.8	42.3	–	88.7	98.2	9.0	–	30.4
	QiMeng-Xpiler w/o SMT	98.2	60.7	–	65.5	97.6	52.4	–	57.1
	QiMeng-Xpiler w/o SMT + Self-Debugging	98.8	62.5	–	66.1	98.2	52.4	–	57.1
	QiMeng-Xpiler	**100**	**100**	–	**100**	**100**	**86.9**	–	**96.4**
C with VNNI	GPT-4 Zero-Shot	57.1	0	60.1	–	8.3	0	8.9	–
	OpenAI o1 Zero-Shot	66.1	0	97.0	–	10.1	0	96.4	–
	GPT-4 Few-Shot	81.5	41.7	74.4	–	14.3	6.0	12.5	–
	OpenAI o1 Few-Shot	87.5	55.4	97.0	–	51.2	10.7	96.4	–
	QiMeng-Xpiler w/o SMT	95.8	78.0	87.5	–	83.9	58.3	85.7	–
	QiMeng-Xpiler w/o SMT + Self-Debugging	97.0	84.5	89.3	–	83.9	58.3	85.7	–
	QiMeng-Xpiler	**100**	**99.4**	**100**	–	**98.2**	**88.7**	**99.4**	–

Table 9: Accuracy comparison to rule-based methods.(%)

Direction	Method	Compilation	Computation
CUDA C → HIP	Hipify	85.7	85.7
	QiMeng-Xpiler	100	100
C → CUDA C	PPCG	47.6	47.6
	QiMeng-Xpiler	100	98.2

from being reliably applied to transcompilers, which demand extremely high accuracy. In contrast, our approach combines LLM-assisted program transformation with SMT-based repairing, enabling automatic program translation with guaranteed correctness. **(3) *QiMeng-Xpiler* performs better than the SOTA rule-based methods** (Table 9). For C → CUDA C, *QiMeng-Xpiler* achieves 100% compilation and 98.2% computation accuracy which is ∼ 50% higher than PPCG. For the easier CUDA C → HIP task, *QiMeng-Xpiler* successfully converts and executes with 100% accuracy, outperforming HIPIFY, which achieves 85.7%. Also, this result shows that *QiMeng-Xpiler*'s flexibility across various DLS without much adaptation cost while rule-based methods cannot.

We conduct an ablation study to further analyze our neural-symbolic synthesis paradigm, shown in Table 8. Without the SMT solver, although *QiMeng-Xpiler* still achieves better accuracy than pure LLM-based methods, it cannot reach 100% accuracy (e.g., 52.4% computation accuracy in HIP → BANG C direction). This situation persists even after incorporating *QiMeng-Xpiler* with the SOTA LLM-based code generation method, Self-Debugging [22]. Due to transcompilers' high demand for accuracy, the methods that cannot achieve or close to 100% accuracy can be hardly applied in practice. In contrast, leveraging SMT-based code repairing can significantly improve the computation accuracy of program transcompilation, achieving a range of 86.9% to 100%. This underscores the necessity of the SMT solver or, more broadly, neural-symbolic synthesis for the transcompiler.

8.2 Evaluations on Execution Performance

We evaluate the performance on the four most common program transcompilation directions, i.e., C with VNNI → CUDA C, CUDA C → BANG C, CUDA C → HIP, and CUDA C → C with VNNI, in Figure 7. Note that we also show the functionally correct cases on each type of operator in the line chart and report the average performance across them. Results show that *QiMeng-Xpiler* achieves an average performance of 0.78× than its manually optimized counterparts across the four transcompilation directions and various operators. This performance gap stems from manually optimized operators can leverage highly specialized techniques tailored to specific hardware, such as handwritten assembly, deeply pipelined execution (e.g., 5-stage pipeline on MLU, multi-stage pipeline on GPU [29]), and aggressive loop unroll on CPU or GPU.

8.3 Case Study: Trancompilation from CUDA C to BANG C

We further demonstrate the superiority and effectiveness of *QiMeng-Xpiler* by comparing the transcompilation results of different methods in the CUDA C → Bang C transcompilation direction. This task is particularly challenging and representative because 1) it involves transcompiling between

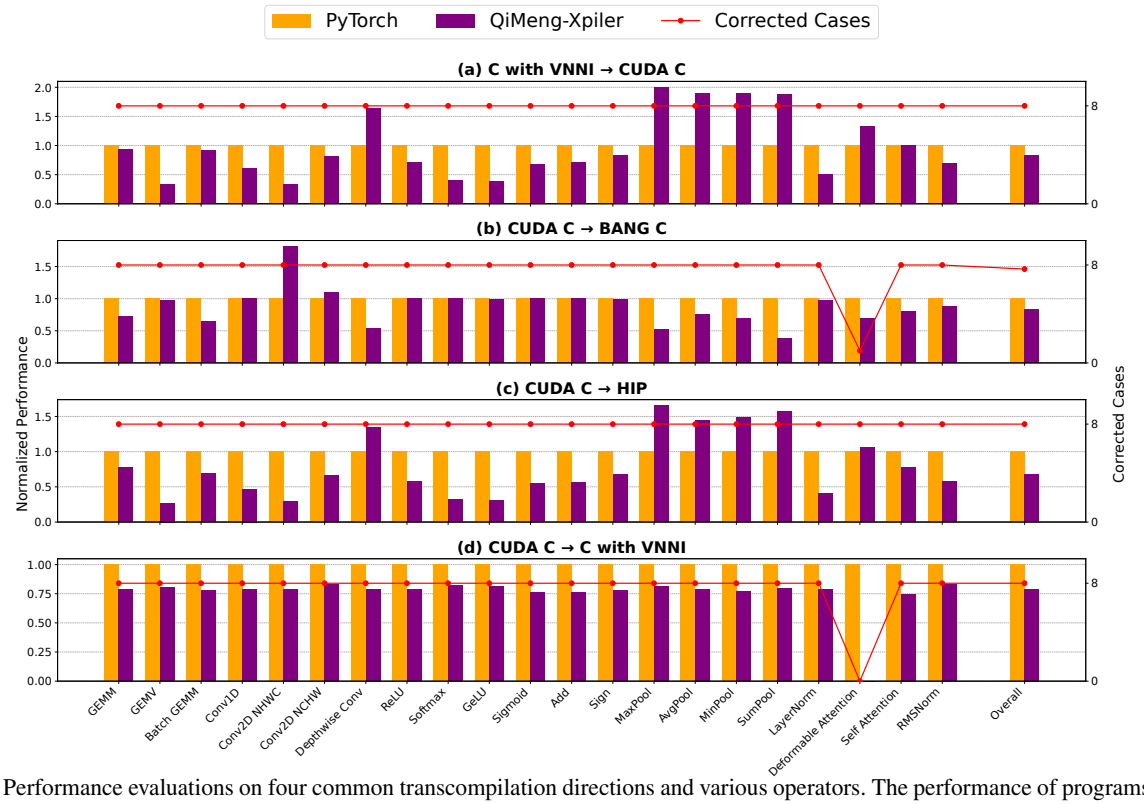


Figure 7: Performance evaluations on four common transcompilation directions and various operators. The performance of programs generated by *QiMeng-Xpiler* is compared to its manually optimized counterparts, PyTorch, with backend libraries such as cuDNN/cuBLAS, CNNL, rocBLAS, and oneDNN.

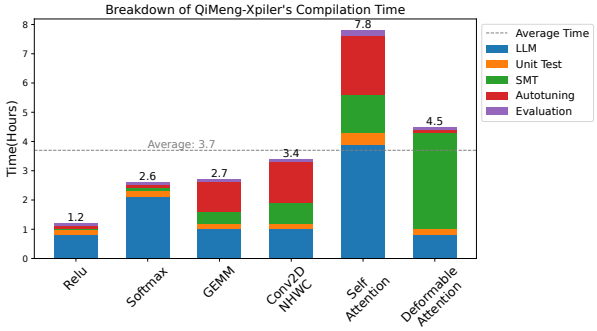


Figure 8: Breakdown of *QiMeng-Xpiler*'s compilation time.

two distinct programming models, namely translating from SIMT programs to SIMD programs, and 2) it targets an uncommon language, which has significantly less training data available for LLMs.

Overall, *QiMeng-Xpiler* achieves much more improvements in compilation/computation accuracy in CUDA C → BANG C compared to other transcompilation directions. While OpenAI o1 Zero-Shot shows excellent performance in some transcompilation directions, such as CUDA C → HIP, with compilation and computation accuracy of 85.7% and 82.7% respectively, its performance in CUDA → Bang translation is notably lacking, with results at 0%. This shortfall is primarily due to the aforementioned two challenges, which exceeds OpenAI o1 Zero-Shot's capabilities.

Although OpenAI o1 Few-Shot manages to boost compilation and computation accuracy substantially from 0% to 50.6% and 0% to 7.7%, respectively, by providing CUDA → Bang translations examples, there still exists a large room attaining the complete accuracy. In contrast, *QiMeng-Xpiler* w/o SMT achieves much higher compilation and computation accuracy of 82.7% and 54.2% correspondingly. This improvement stems from two key approaches adopted in *QiMeng-Xpiler* w/o SMT : 1) it gains a comprehensive understanding of the target language's keywords, syntax, and programming APIs by automatically annotating programs via document retrieval, thus compensating for the lack of domain-specific knowledge, and 2) whereas typical transcompilers attempt intricate inter-language translations in a single synthesis step, *QiMeng-Xpiler* w/o SMT splits the entire process into a series of LLM-based transformation passes. This enables *QiMeng-Xpiler* w/o SMT to effectively facilitate smooth translations between two substantially different source and target languages. Crucially, with the integration of SMT methods, *QiMeng-Xpiler* attains remarkable accuracy in the CUDA C → BANG C translation, achieving 100% in compilation and 91.7% in computation. This accuracy is largely due to *QiMeng-Xpiler*'s implementation of small-scale symbolic synthesis, which ensures the functional equivalence of each transformation pass guided by the LLM. In conclusion, our experiments prove that *QiMeng-Xpiler* successfully ac-

complishes transcompiler tasks across different DLS, even for a new and relatively uncommon DLS.

8.4 Compilation Time

Figure 8 shows the compilation time of 6 typical operators when translating from CUDA C to BANG C, which ranges from 1.2 to 7.8 hours, with 3.7 hours on average. We further analyze the breakdown of *QiMeng-Xpiler*'s compilation time and conclude that: 1) SMT is only triggered when the LLM fails to produce a correct translation, so for simpler programs, the proportion of time spent on SMT is smaller; 2) For matrix multiplication-like operators with a larger search space, the proportion of time spent on autotuning significantly increases; 3) Compilation time increases as the number of special intrinsics used in the program grows.

8.5 Productivity Improvement

We further evaluate how *QiMeng-Xpiler* improves programming productivity in real-world scenarios on 2 representative DLS, i.e., GPU and MLU. Specifically, we compare the development costs and performance achieved by manually implementing tensor programs versus those generated by *QiMeng-Xpiler* through automatic transcompilation, on the most challenging operator, Deformable Attention, with $\sim$ 200 LoCs.

We invited two CS master's students and two software engineers as junior and senior coders, respectively. Note that performance is normalized against the manually implemented programs by the senior coders. Table 10 shows that, programming productivity is improved by 34.3$\times$ for GPU and 96.0$\times$ for MLU via transcompiling legacy tensor program. Although *QiMeng-Xpiler* fails to automatically generate a functional-correct program for MLU, both junior and senior coders could debug the program with minimal effort, requiring an additional 3 hours and 0.5 hours, respectively.

Table 10: Productivity Improvement by *QiMeng-Xpiler*.

Deformable Attention ($\sim$ 200LoCs)		**CUDA C-> BANG C**		**C with VNNI -> CUDA C**	
		Costs	Performance	Costs	Performance
Senior Coder	Manual	$\sim$6 d	100%	$\sim$1 d	100%
	Ours	4.5 + 0.5 h	69.20%	2.1 h	132.50 %
	Time Saving	$\sim$28.8$\times$		$\sim$11.4$\times$	
Junior Coder	Manual	$\sim$30 d	49.85%	$\sim$3 d	75.76%
	Ours	4.5 + 3 h	65.17%	2.1 h	132.50 %
	Time Saving	$\sim$**96.0**$\times$		$\sim$**34.3**$\times$	

8.6 Case Study on FlashAttention

To evaluate *QiMeng-Xpiler*'s effectiveness on real-world, performance-critical kernels, we conducted a case study on FlashAttention (FA1) [24] and FlashAttention-2 (FA2) [23], a memory-efficient attention algorithm widely used in LLMs. Note that FlashAttention kernels require fine-grained parallelism and high-throughput shared memory access, capabilities that are not efficiently supported by CPUs. Table 11 shows the normalized performance of *QiMeng-Xpiler* on FA1 and FA2 compared to vendor-optimized implementations on different transcompilation directions. Results show that *QiMeng-Xpiler* achieves 0.61$\times$-0.81$\times$ the performance of its native implementations on all transcompilation directions, primarily due to its limited ability to match the intricate shared-memory tiling and data movement strategies employed in hand-optimized implementations.

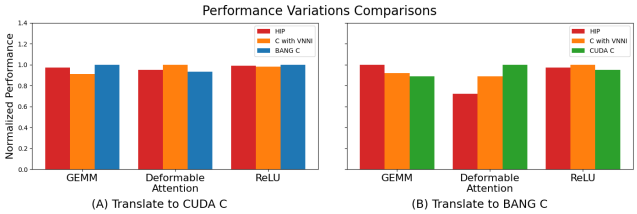


Figure 9: Normalized performance of `GEMM`, `Deformable Attention`, and `ReLU` across different trancompilation directions.

Table 11: Normalized performance of *QiMeng-Xpiler* on FA1 and FA2 compared to vendor optimized implementations. **Configuration:** Batch Size = 8, Number of Heads = 16, Head Dimension = 32, Sequence Length = 1024.

Source	**Operator**	**Target**		
		HIP	**BANG C**	**CUDA C**
HIP	FA1	–	0.74	0.71
	FA2	–	0.76	0.74
BANG C	FA1	0.77	–	0.75
	FA2	0.78	–	0.61
CUDA C	FA1	0.69	0.70	–
	FA2	0.81	0.63	–

8.7 Performance Variations Comparisons

In this experiment, we evaluate how *QiMeng-Xpiler*'s performance varies across different source platforms when targeting the same target platform. Figure 9 shows the normalized performance of three representative operators, `GEMM`, `Deformable Attention`, and `ReLU`, when transcompiled to CUDA C (left) and BANG C (right) from various source platforms. Results show that the transcompiled programs achieve comparable performance because *QiMeng-Xpiler* first converts all source programs into a unified intermediate representation (e.g., scalar C code), decoupling subsequent optimizations from source-specific tuning. The only exception is the transcompilation of Deformable Attention from HIP to BANG C, which achieves only 72% performance. This is mainly because a conditional statement in the HIP source code hinders optimization during conversion, preventing the generated BANG C program from being effectively optimized.

8.8 Failure Case

Since *QiMeng-Xpiler* cannot guarantee 100% correct translation in all cases, we analyzed the failure cases in depth and summarized key limitations as directions for future work.

Complex control flow. When the input program involves complex control structures like multiple nested loops and con-

ditionals, both LLMs and SMT tools struggle to capture the correct parallel semantics, resulting in translation or repair failures. For example, when transcompiling Deformable Attention from CUDA C to BANG C (as shown in Figure 10), the code snippet involves highly complex control flow. As a result, the neural component (i.e., LLMs) of *QiMeng-Xpiler* fails to generate the required SIMD intrinsics for BANG C, and the symbolic component (i.e., SMT solver) cannot infer the mathematical constraints needed for repair.

Difficult-to-understand computations. Although GPT-4 can recognize a wide range of scalar computations, it struggles to comprehend arbitrary special instructions, hindering its ability for accurate annotation.

To address these limitations, we plan to incorporate more advanced LLMs and SMT techniques, which are expected to significantly enhance *QiMeng-Xpiler*'s capability in handling complex, real-world deep learning kernels.

```
for (int32_t i_m_5 = 0; i_m_5 < 8; ++i_m_5) {
  if ((((((int32_t *)xy_rounded)[(i_m_5 + 8)] < 0) || (((int32_t *)height_width)[0] <= ((int32_t *)xy_rounded)[(i_m_5 + 8)])) ||
    ((((int32_t *)xy_rounded)[(i_m_5 + 24)] < 0)) || (((int32_t *)height_width)[1] <= ((int32_t *)xy_rounded)[(i_m_5 + 24)]))) {
    for (int32_t i_d_7 = 0; i_d_7 < 512; ++i_d_7) {
      ((float *)corner_values)[(((i_m_5 * 512) + i_d_7) + 12288)] = 0.000000e+00f;
    }
  }
}
```

Figure 10: The complex control flow in `Deformable Attention`.

9 Related Work

Rule-based approaches. These methods focus on transcompiling source programs to target languages using expert-defined rules [5, 7, 13, 50]. For instance, C2Rust [13] and CxGo [5] translate C code to Rust and Go, respectively, while HIPIFY [7] converts CUDA to HIP. PPCG [50] extracts the polyhedral model from the source program, applies predefined rules for scheduling and parallelization, and generates code from the transformed model. These approaches are labor-intensive, relying on manual rules that require extensive knowledge of both the source and target languages, limiting their applicability to specific platforms. Moreover, they often struggle with irregular code structures, reducing their robustness and general applicability.

Data-driven approaches. Data-driven methods, which train neural networks using supervised or unsupervised corpora, have gained prominence following the success of Neural Machine Translation (NMT). Tools like TransCoder [43] use sequence-to-sequence models for translating between C++, Java, and Python. More recent works leverage LLMs (e.g., CodeX [21], StarCoder [34], CodeGen [38], CodeT5 [51], CodeGeeX [55], LLaMA [47], Gemini [46], GPT-4 [15], and OpenAI o1 [8]) for program translation, achieving superior performance. However, while these methods reduce human effort, they often fail to guarantee the functional correctness of the translated code, especially across different DLS.

Symbolic synthesis approaches. Symbolic synthesis approaches generate semantically equivalent code from input-output pairs or formal semantic specifications. For example, C2TACO [36] uses a guided enumerative synthesizer and automatically generated I/O examples to translate C tensor code into the TACO DSL [33]. Recent methods like MetaLift [18] and Tenspiler [42] adopt a more extensible approach by first converting the source program into a unified intermediate representation (IR) and then synthesizing target programs in the same IR domain. These approaches reduce the need for manual transformation rules and are effective for small-scale projects. However, they rely on expensive search-based SMT solvers, which makes them less scalable for larger, real-world applications. Additionally, methods like MetaLift and Tenspiler require users to manually define the semantics of the target language using the specification IR, which is both challenging and error-prone.

In summary, existing approaches either demand significant manual effort, suffer from functional correctness issues, or have limited scalability, making them ineffective for automatically translating programs across DLS with different programming models. In contrast, *QiMeng-Xpiler* is the first work to tackle this challenge by proposing a novel neural-symbolic synthesis framework, where translation is performed through a series of LLM-assisted transformations, with functional equivalence ensured for each transformation via small-scale symbolic synthesis.

10 Conclusion

We propose a neural-symbolic synthesis approach, *QiMeng-Xpiler*, for automatically translating tensor programs across heterogeneous DLS with different programming models. *QiMeng-Xpiler* conducts the program translation as a series of LLM-assisted transformation passes, where incorrect code snippets are repaired by small-scale symbolic synthesis. In addition to ensuring functional correctness, *QiMeng-Xpiler* also employs hierarchical auto-tuning to improve the performance of translated programs. Experimental results on 4 different DLS demonstrate that *QiMeng-Xpiler* correctly translates different tensor programs at the accuracy of 95% on average, and the performance of the translated program achieves an average of 0.78× of vendor-provided manually-optimized libraries. Moreover, the programming productivity of DLS is improved by up to 96.0×.

Acknowledgments

We would like to extend our most sincere gratitude to our shepherd, Shan Lu, and the reviewers for their feedback and suggestions. This work is partially supported by Strategic Priority Research Program of the Chinese Academy of Sciences (Grants No.XDB0660300, XDB0660301, XDB0660302), Science and Technology Major Special Program of Jiangsu (Grants No. BG2024028), the NSF of China (Grants No.U22A2028, 62302483, 6240073476, 62302482, 62302478), CAS Project for Young Scientists in Basic Research (YSBR-029) and Youth Innovation Promotion Association CAS.

References

[1] Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas.

[2] Cambricon BANG C Developer Guide. https://www.cambricon.com/docs/sdk_1.13.0/cntoolkit_3.5.2/cambricon_bang_c_4.5.1/index.html.

[3] Cambricon MLU. https://www.cambricon.com/.

[4] Cloud Computing Services - Amazon Web Services (AWS). https://aws.amazon.com/.

[5] cxgo. https://github.com/gotranspile/cxgo.

[6] Google Cloud: Cloud Computing Services. https://cloud.google.com/.

[7] HIPIFY. https://github.com/ROCm/HIPIFY.

[8] Introducing OpenAI o1. https://openai.com/o1/.

[9] IPU Processors. https://www.graphcore.ai/products/ipu.

[10] Microsoft Azure: Cloud Computing Services. https://azure.microsoft.com/.

[11] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[12] NVIDIA Tensor Core. https://www.nvidia.cn/data-center/tensor-cores/.

[13] C2Rust, [n.d]. https://github.com/immunant/c2rust.

[14] oneAPI Deep Neural Network Library (oneDNN), [n.d]. https://github.com/intel/mkl-dnn.

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

[16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[17] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. Mosaic: An interoperable compiler for tensor algebra. *Proceedings of the ACM on Programming Languages*, 7(PLDI):394–419, 2023.

[18] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[19] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[20] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[22] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[23] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[24] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[26] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. Cox: Exposing cuda warp-level functions to cpus. *ACM Trans. Archit. Code Optim.*, 19(4), sep 2022.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[28] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[29] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems*, 5:680–694, 2023.

[30] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 703–718, 2022.

[31] Alister Johnson, Camille Coti, Allen D Malony, and Johannes Doerfert. Martini: The little match and replace tool for automatic application rewriting with code examples. In *European Conference on Parallel Processing*, pages 19–34. Springer, 2022.

[32] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[33] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 943–948. IEEE, 2017.

[34] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[35] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[36] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael FP O'Boyle. C2taco: Lifting tensor code to taco. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 42–56, 2023.

[37] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *In Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, ASPLOS '21, pages 337–340, 2008.

[38] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[39] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[40] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Efficient compilation of cuda kernels for high-performance computing on fpgas. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–26, 2013.

[41] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah E. Chasins, and Rastislav Bodík. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In Michael F. P. O'Boyle and Keshav Pingali, editors, *In Proceedings of International Conference on Programming Language and Design Implementation (PLDI)*, pages 396–407, 2014.

[42] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Tenspiler: A verified-lifting-based compiler for tensor operations. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*, volume 313 of *LIPIcs*, pages 32:1–32:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[43] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc., 2020.

[44] Harshil Shah and David Barber. Generative neural machine translation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[45] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, May 2015.

[46] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[47] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[48] Andrew Trotman, Antti Puurula, and Blake Burgess. Improvements to bm25 and language models examined. In *Proceedings of the 19th Australasian Document Computing Symposium*, ADCS '14, page 58–65, New York, NY, USA, 2014. Association for Computing Machinery.

[49] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation extended abstract.

[50] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4), jan 2013.

[51] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[52] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F. P. O'Boyle. Bind the gap: Compiling real software to hardware fft accelerators. In *In Proceedings of International Conference on Programming Language and Design Implementation (PLDI)*, page 687–702, 2022.

[53] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[54] Zhuofan Xia, Xuran Pan, Shiji Song, Li Erran Li, and Gao Huang. Vision transformer with deformable attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4794–4803, 2022.

[55] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humanevalx. *arXiv preprint arXiv:2303.17568*, 2023.

WaferLLM: Large Language Model Inference at Wafer Scale

Congjie He[1] Yeqi Huang[1] Pei Mu[1] Ziming Miao[2] Jilong Xue[2] Lingxiao Ma[2] Fan Yang[2] Luo Mai[1]

[1]*University of Edinburgh* [2]*Microsoft Research*

Abstract

Emerging AI accelerators increasingly adopt wafer-scale manufacturing technologies, integrating hundreds of thousands of AI cores in a mesh architecture with large distributed on-chip memory (tens of GB in total) and ultra-high on-chip memory bandwidth (tens of PB/s). However, current LLM inference systems, optimized for shared memory architectures like GPUs, fail to exploit these accelerators fully.

We introduce WaferLLM, the first wafer-scale LLM inference system. WaferLLM is guided by a novel PLMR model (pronounced as "Plummer") that captures the unique hardware characteristics of wafer-scale architectures. Leveraging this model, WaferLLM pioneers wafer-scale LLM parallelism, optimizing the utilization of hundreds of thousands of on-chip cores. It also introduces MeshGEMM and MeshGEMV, the first GEMM and GEMV implementations designed to scale effectively on wafer-scale accelerators.

Evaluations show that WaferLLM achieves up to 200× higher accelerator utilization than state-of-the-art methods. Leveraging a wafer-scale accelerator (Cerebras WSE2), WaferLLM delivers GEMV operations 606× faster and 16× more energy-efficient than on an NVIDIA A100 GPU. For full LLM inference, WaferLLM achieves 10-20× speedups over A100 GPU clusters running SGLang and vLLM. These advantages are expected to grow as wafer-scale AI models, software, and hardware continue to mature. WaferLLM is open-sourced at https://github.com/MeshInfra/WaferLLM.

1 Introduction

Large Language Model (LLM) inference is a rapidly growing workload. It has two phases [15]: (i) the *prefill phase*, which processes input tokens (the prompt) and spends most of its cycles on General Matrix Multiply (GEMM); and (ii) the *decode phase*, which generates tokens one by one in an autoregressive manner, primarily performing General Matrix-Vector Product (GEMV). Decode requires repeatedly loading the entire LLM model into on-chip memory, with GEMV dominating its cycles. Since LLMs generate many tokens, especially in the test-time scaling scenario, such as the OpenAI-o1/o3 [16, 33] and DeepSeek-R1 [13], inference is constrained by GEMV latency, making it inherently memory-bandwidth-bound.

To address memory bandwidth bottlenecks, AI accelerators are increasingly adopting system-on-wafer integration [21]. This approach scales chip area to a full wafer, up to 100× larger than a typical GPU die, enabling significantly more on-chip cores, memory, and bandwidth. Examples include Cerebras WSE [24] and upcoming Tesla Dojo [41]. The Cerebras WSE-2, for instance, integrates 850,000 cores with 40GB of on-chip memory, 1,000× more than GPUs, and provides 22PB/s memory bandwidth, 7,000× higher than GPUs. TSMC anticipates widespread adoption of system-on-wafer integration, citing advantages in performance, energy-efficient die-to-die communication, and cost reduction. IEEE similarly forecasts a surge in wafer-scale computing by 2027 [21]. Wafer-scale accelerators are already seeing real-world deployment, particularly in model serving. In February 2025, Mixtral and Perplexity adopted wafer-scale chips, achieving cost parity with GPUs in terms of tokens per dollar [40, 44]. G42 now operates data centers fully outfitted with wafer-scale accelerators, and Cerebras has secured major commercial contracts [39].

Unlocking the potential of wafer-scale accelerators is challenging because current LLM systems rely on *shared memory architectures* typical of GPUs and TPUs. Wafer-scale accelerators, however, adopt *network-on-chip* (NoC) designs that interconnect millions of cores with local memory in a *massive-scale, mesh-based memory architecture*. This architecture far exceeds the scale of on-chip crossbars (e.g., one-hop NUMA such as GraphCore IPU), multi-socket NUMA [2], and high-density AI clusters (hundreds of GPUs per pod) [17]. Without fully addressing this fundamental shift in memory architecture, directly applying designs from state-of-the-art systems like T10 [25] and Ladder [45] to wafer-scale devices often results in extremely poor performance.

To address these challenges, we propose a *device model* that captures the critical hardware properties of wafer-scale accelerators, highlighting key differences from shared-memory devices. This model enables us to evaluate current LLM inference design principles, identify non-compliant areas, and pinpoint where new approaches are required. Guided by this model, we can achieve an ambitious system design: *running complete LLM inference on a single chip*, minimizing costly off-chip communication and maximizing on-chip memory bandwidth utilization.

The above idea motivates WaferLLM, the first wafer-scale LLM inference system, yielding several contributions:

(1) Device model for wafer-scale accelerators. We propose

the PLMR model[1], which captures the key hardware properties of wafer-scale accelerators: (i) Massive **P**arallel cores (P): Millions of cores can be integrated on a large wafer, requiring systems to effectively partition LLMs and their operations. (ii) Highly non-uniform memory access **L**atency (L): Inter-core data access exhibits significant variation, with latency differences up to 1,000×, necessitating the system to mitigate this. (iii) Constrained per-core local **M**emory (M): Each core has limited memory (tens of KBs to several MBs), requiring efficient memory usage. (iv) Limited hardware-assisted **R**outing (R): The NoC routing hardware is constrained by the area size per core and can only support a limited number of routing paths, e.g., less than 2^5 on Cerebras WSE-2.

(2) Wafer-scale LLM parallelism. We propose an effective, PLMR-compliant LLM parallelism policy for wafer-scale accelerators. In the prefill phase, we design fine-grained partitioning to achieve million-core parallelism. For the decode phase, where tensor dimensions are insufficient for partitioning, we design fine-grained replication to enable parallelism with minimal communication costs. As a result, WaferLLM achieves larger-scale and finer-grained parallelism (satisfying P in PLMR) than GPU-based approaches. Additionally, we replace conventional GPU-based GEMM and GEMV operators with new algorithm designed for the PLMR model (satisfying L, M, and R) and propose tensor placement strategies that eliminate matrix transpositions, which are costly with a mesh NoC (satisfying L).

We also designed a scalable KV-cache management method for wafer-scale devices. This approach features a novel KV cache shift method to ensure balanced core usage (satisfying P and M), avoiding skewed utilization of cores caused by KV cache concatenation methods common on GPUs.

(3) Wafer-scale GEMM. We propose MeshGEMM, a scalable GEMM algorithm for wafer-scale devices, accelerating the prefill phase. Unlike conventional distributed GEMM algorithms, MeshGEMM achieves full PLMR compliance by leveraging two key operations: *cyclic shifting* and *interleaving*. Cyclic shifting ensures algorithm correctness while maintaining bounded usage of local memory (satisfying M). The interleaving operation minimizes communication latency in the mesh NoC, effectively reducing the overhead of highly non-uniform memory latency (satisfying L) and routing resources (satisfying R).

(4) Wafer-scale GEMV. We propose MeshGEMV, a scalable GEMV algorithm for wafer-scale devices, accelerating the decode phase. Unlike existing GEMV implementations, MeshGEMV uses a novel *K-tree allreduce* algorithm to aggregate local GEMV results across massive cores. This algorithm ensures routing resource usage meets the hardware's limitation (satisfying R) and reduces communication latency (satisfying L).

[1]PLMR model can be pronounced as "Plummer"

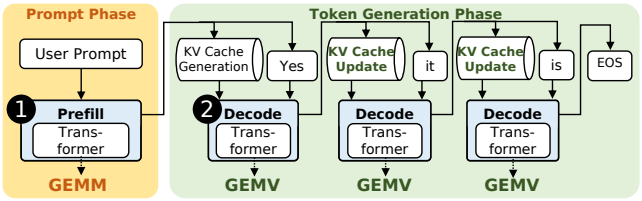


Figure 1: Key components in LLM inference

We implemented WaferLLM on the Cerebras WSE engine using approximately 7,000 lines of CSL (a C-like programming language) for LLM parallelism, MeshGEMM, and MeshGEMV, and 2,000 lines of Python for loading LLM checkpoints, launching inference, and execution parallelism policy.

We conducted end-to-end LLM inference experiments with various models, including full LLaMA3-8B and LLaMA2-13B, as well as subsets of layers of CodeLLaMA-34B, and QWen2-72B. By combining wafer-scale LLM parallelism, GEMM and GEMV, WaferLLM outperforms state-of-the-art (SOTA) systems: (i) 100-200× faster than T10 [25], the SOTA system for massive cores with a distributed on-chip memory architecture, and (ii) 200-400× faster than Ladder [45], the SOTA system for shared-memory architectures.

Micro-benchmarks further show that MeshGEMM is 2-3× faster than SUMMA [42], the default optimized GEMM for Cerebras WSE, and Cannon [6], the SOTA GEMM for supercomputers with large-scale mesh architectures. MeshGEMV achieves 4-8× speedups over Cerebras's optimized GEMV, and 606× than a single A100 GPU. Additionally, WaferLLM 's cache shift method is up to 400× more scalable than the KV cache SOTA on GPUs, such as PagedAttention [20].

Combined, WaferLLM (on Cerebras WSE-2) outperforms SGLang (on single A100) by 30-40×. Compared to the optimal performance of SGLang on A100 multi-GPUs connected with NVLink and RDMA, WaferLLM delivers a 10-20× faster end-to-end speed and is 2.5× more energy-efficient. The reduced gains from GEMV to LLM are due to current limitations in software, hardware, and existing LLM model designs. We anticipate stronger performance as wafer-scale AI computing matures and these limitations are addressed.

2 Background and Motivation

2.1 LLM inference and its key constraint

An LLM inference system typically performs auto-regressive token-by-token generation, as illustrated in Figure 1. The model comprises multiple transformer layers, dominated by self-attention and feedforward blocks. Inference operates in two phases: prefill and decode. The total cycles of the prefill phase are dominated by GEMM operations (shown by ❶). While the total cycles of the decode phase are dominated by

	System-on-Die	System-on-Wafer
Area (mm^2)	Up to 858	Up to 73062
# Transistors (TSMC n3)	~1s Tillion	~10s Trillion
# Cores	1,000s-10,000s	100,000s-1,000,000s
On-Chip Memory	10s-100s MB	~10s GB
Memory Bandwidth	1s TB/s	~10s PBs/s
Attached HBM	~10s-100s GB	10s TB (via TSMC SoW) [21]
Die-to-Die Bandwidth (TB/s)	~1s-10s (via off-chip)	~10s-100s (via on-chip)
Die-to-Die Latency (ns)	~100s (via off-chip)	~1s (via on-chip)
Die-to-Die Power (pJ/bit)	~10s (via off-chip)	~0.1s (via on-chip)

Table 1: System-on-Die vs. System-on-Wafer

GEMV operations (shown by ❷).

LLM inference is memory bandwidth-bound. Model weights (10-1,000 GB) are fetched repeatedly from external memory during inference, as GPUs typically have only 100 MB of on-chip memory. For per request, generating thousands of tokens per second demands hundreds of TB/s bandwidth, far exceeding the capabilities of HBM (high bandwidth memory) on current GPUs.

While tensor parallelism across GPUs can increase bandwidth, mitigating communication bottlenecks in a large GPU cluster remains challenging. Also, adding GPUs improves throughput for concurrent queries but does not reduce time per output token (TPOT), as each query is still memory bandwidth-limited.

2.2 Reasons for wafer-scale accelerators

To increase memory bandwidth, accelerator designers are increasingly adopting system-on-wafer integration [21] for several reasons:

Performance advantages. System-on-wafer technology allows trillions of transistors to be integrated into a single wafer-scale chip, up to 100× more than a typical GPU die, shown in Table 1. This enables millions of AI-optimized cores, providing tens of GBs of on-chip memory and up to tens of PB/s memory bandwidth, 1,000× higher than a standard GPU's several TB/s. Future wafer-scale chips can also attach 40-80× more HBM chips to their edge compared to a standard die [21].

Integration efficiency. System-on-wafer excels at integrating massive parallel cores, with wafer-based die-to-die connections offering up to 10× more bandwidth per unit area, 100-300× latency benefit, and nearly 100× better power efficiency per bit than conventional PCB-based chip-to-chip interconnection [24] (e.g., NVIDIA NVLink, PCIe), shown in Table 1. As noted earlier, LLM inference is primarily constrained by memory bandwidth due to intensive data access. In distributed settings, inter-chip communication overhead from remote memory access, especially during decode-phase GEMV, limits scalability. Thus, system-on-wafer integration offers lower-latency and high-bandwidth interconnects and improved efficiency over conventional chip-to-chip links.

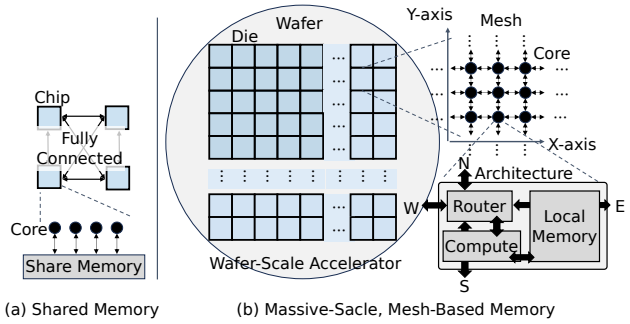


Figure 2: Massive-scale mesh-based memory architecture

Lower cost. Wafer-scale integration can lower the manufacturing cost, since a significant fraction of the cost of fabrication (typically 30-50%) is related to testing and packaging the individual chips [48]. Additionally, wafer-scale integration has made notable progress in yield improvement. Companies such as TSMC are also developing techniques to integrate fully tested dies on a single wafer, further enhancing yield.

2.3 Challenges for wafer-scale LLM inference

The key challenge in leveraging wafer-scale accelerators for LLM inference is their shift to a distributed, non-uniform memory architecture on a single chip. Current LLM systems are optimized for shared memory (single chip) or fully connected architectures (e.g., GPU pods), as shown in Figure 2(a). However, as on-chip memory size grows, these architectures face exponential manufacturing costs and performance degradation, driving the need for a distributed on-chip architecture.

AI accelerator designers predominantly use a **mesh-like network-on-chip (NoC)** to connect **massive cores** (ranging from hundreds of thousands to millions), as shown in Figure 2(b). The mesh topology is favored for its efficiency in core arrangement, enabling effective cooling [29], power delivery [19], and cost-efficient wiring [38, 46], with each core communicating only with nearby neighbors, as shown in Figure 2(b). Alternative topologies, such as 3D torus or tree structures, are impractical due to high on-chip wiring costs. Therefore, wafer-scale chip makers such as Cerebras WSE [24] and Tesla Dojo [41] adopt massive-scale mesh architectures. Even non-wafer-scale accelerators such as Meta MTIA [30], Tenstorrent [18], and others [4, 31] use mesh to scale cores on a chip.

The massive-scale mesh architecture presents challenges for several LLM operations due to their high data movement demands: (i) managing LLM models and KV cache, (ii) GEMM operations during the prefill phase, and (iii) GEMV operations during the decode phase. Other operations, such as element-wise computations such as dot-product and activation functions, require no data movement and naturally benefit from parallelism. Operations needing allreduce, such as RMSNorm and Softmax, can leverage GEMV solutions.

3 Device Model for Wafer-Scale Accelerators

3.1 The PLMR model

We develop the PLMR model to capture the unique hardware properties of wafer-scale accelerators and to motivate system requirements needed for utilizing this emerging hardware.

(1) **Massive Parallelism (P)**: A wafer-scale accelerator can accommodate millions of parallel cores, compared to thousands in GPUs. Each core features a local hardware pipeline that overlaps data ingress, egress, computation, and memory access at the cycle level. This requires the computation to be partitioned at a massive scale and a fine-grained schedule to overlap computation, memory access, and NoC communication.

(2) **Highly non-uniform memory access Latency (L)**: Accessing memory across cores in a mesh incurs highly non-uniform latency. In a mesh with $N_w \times N_h$ cores, organized as a rectangle, the maximum number of NoC hops between two cores is $N_w + N_h$, and the worst-case memory access latency is given by $\alpha(N_w + N_h) + \beta r$, where $r < N_w + N_h$ denotes the number of routing stages along the communication path. Here, α denotes the per-hop transmission latency, the cost incurred when a message is directly forwarded at a core according to the router hardware's pre-configured rules, increasing with hops. β represents the per-routing latency, the overhead when a message is involved in header parsing and rewriting by software at a core when forwarding [27]. Typically, $\alpha < \beta$. In a mesh with a million cores, the maximum number of hops and routing can reach up to several thousand, resulting in up to a thousand times latency gap between local and remote memory access. Consequently, minimizing long-range communication is critical for performance.

(3) **Constrained per-core local Memory (M)**: Each core has a small local memory (tens of KBs to several MBs), as performance and energy efficiency decline with larger capacities [47]. As a result, computation data must be explicitly partitioned into fine-grained chunks to fully fit within the constraints of each core's local memory.

(4) **Constrained Routing resources (R)**: Wafer-scale accelerators integrating millions of cores impose strict constraints on each core's routing circuit complexity or routing table size. For example, on the Cerebras WSE-2, each core can recognize only message headers with a 5-bit address code. Consequently, each core can support at most 2^5 distinct routing paths, and the software system must carefully plan these paths. For long-distance remote communication, a pair of cores can consume routing resources to establish a direct routing path, incurring only α latency. However, if the number of routing paths exceeds hardware limits, messages must be relayed through multiple intermediate cores, introducing additional β latency.

We expect these properties to remain relevant, as they are rooted in the fundamental characteristics of hardware and its manufacturing process. The PLMR model applies to both current (Cerebras WSE) and future (Tesla Dojo) wafer-scale devices. Even some non-wafer-scale devices with mesh-based NoC architectures, such as Tenstorrent Blackhole [18], can be represented by PLMR with adjusted parameters for parallelism (P), the size of the mesh (L), or relaxed constraints on local memory (M) and routing resources (R).

3.2 Limitations of state-of-the-art approaches

Leveraging the PLMR model, we analyze why existing AI systems fail to fully utilize wafer-scale accelerators. To run an LLM on a wafer-scale accelerator, we generally have two choices: (i) abstract the distributed local memory in each core as a shared memory and directly access data placed in a remote core through NoC; and (ii) explicitly partition computation into distributed cores and use message passing to exchange necessary data. We analyze two types of representative systems: LLM runtime or DNN compilers for shared memory architecture such as GPUs, e.g., Ladder [45]; and the SOTA compiler for distributed on-chip memory architectures, e.g., T10 [25] for GraphCore IPU.

Shared-memory system. A shared-memory-based DNN compiler such as Ladder usually assumes a uniform memory access pattern within the underlying memory hierarchy, which cannot tolerate the thousands of times latency variance in wafer-scale accelerators when accessing data from remote memory (failing in L). Moreover, these compilers [10, 28, 37, 45, 51, 54, 57] often focus primarily on partitioning computation, with less emphasis on optimizing data partitioning. This approach can easily lead to significant data duplication and violate the memory constraint requirements (failing in M). Finally, these compilers are unaware of the communication distance of each core, poorly addressing the constraint of routing resources.

Distributed-memory system. The T10 system [25] is designed for AI accelerators with an on-chip crossbar, which ensures a constant latency of memory access to any other cores on the same chip. T10 handles small local memory and balances communication loads, addressing memory constraints (M) and routing resource limitations (R). However, on a PLMR device, it fails to account for varying memory access latency (failing in L) and scales to thousands, not millions, of cores (failing in P).

4 Wafer-Scale LLM Parallelism

We present wafer-scale LLM parallelism, featuring new designs across prefill, decode, and KV cache management.

4.1 Prefill parallelism

The parallelism for LLM prefill must ensure compliance with the PLMR model. Key challenges include: (i) Handling multiple large matrices during prefill, requiring effective dimension

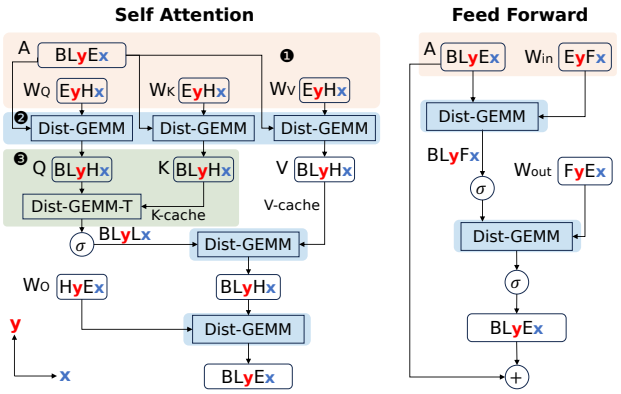


Figure 3: Prefill parallelism plan. E_xF_y represents a matrix of shape EF, where the E dimension is partitioned along the x-axis of cores, and F along the y-axis of cores on a mesh.

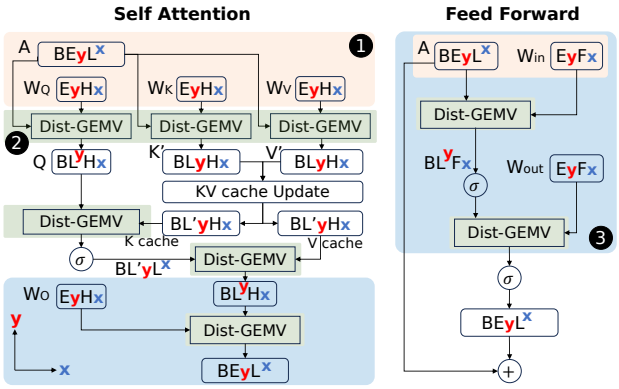


Figure 4: Decode parallelism plan. E^yF_x indicates the E dimension is replicated along the y-axis, and F is partitioned along the x-axis.

partitioning to achieve million-core parallelism (P); (ii) Optimizing GEMM operations, which involve further partitioning and overlapping computation and communication, to minimize long-range communication latency (L), respect local memory constraints (M), and account for limited routing resources (R); and (iii) Handling matrix transposes, which are costly on a mesh NoC (L) but often required for sequential GEMM operations.

Designing fine-grained partitioning for million-core parallelism. To achieve high chip utilization, we propose partitioning two dimensions of the input activation and weight matrices along both the X- and Y-axes of cores. This approach enables finer-grained, million-scale parallelism compared to existing methods [12,15,32,35], which typically partition only the embedding dimension, resulting in insufficient parallelism on PLMR devices.

We illustrate this partitioning using self-attention and feed-forward, as shown in Figure 3. For this discussion, we define the following annotations: the input activation A and weight W are multi-dimensional tensors during the prefill process. B represents the batch size, L the sequence dimension [2], E the embedding dimension, H the head dimension, and F the hidden dimension in the feedforward block. As shown by ❶, the partitioning layout of A is represented as BL_yE_x, where the L dimension is partitioned along the Y-axis of cores, and the E dimension along the X-axis of cores. Similarly, all weight matrices (W_Q, W_K, W_V, W_{in}, and W_{out}) are partitioned across both dimensions.

Designing PLMR-compliant distributed GEMM. We propose replacing conventional GEMM operators, which are designed for shared memory architectures, with a newly designed PLMR-compliant distributed GEMM during the prefill phase (as shown in ❷ of Figure 3). Unlike TPU and GPU systems that primarily rely on allgather operations for GEMM, PLMR-compliant distributed GEMM algorithms achieve high NoC bandwidth utilization while respecting local memory and routing constraints, ensuring compliance with the L, M, and R properties. This PLMR-compliant distributed GEMM is fully described in Section 5.

Using transposed distributed GEMM to avoid matrix transpose. We propose a transpose-free parallelism plan for prefill to avoid matrix transpose, a common operation in LLM systems designed for shared memory architectures. The L property in PLMR highlights that matrix transposition is particularly costly on a wafer-scale device. It requires a core on one corner of the mesh to send data to the opposite diagonal corner, creating a long-range communication path.

Our transpose-free parallelism plan leverages transposed distributed GEMM (denoted as dist-GEMM-T) [11, 42] to compute $Q@K^T$ during LLM prefill, as shown by ❸ in Figure 3. Specifically, the intermediate Q and K tensors, generated by multiplying X with W_Q and W_K, require transposing K before proceeding with dist-GEMM operations due to the on-chip partition shape.

4.2 Decode parallelism

The parallelism strategy for LLM decode must address its memory-bandwidth-intensive nature, presenting several challenges: (i) Decode uses smaller matrices than prefill due to limited input sequences and batch sizes, requiring careful parallelization when dimensions are insufficient for partitioning; (ii) The phase heavily relies on GEMV operations, which are less compute-intensive than GEMM, resulting in short computation phases with limited overlap with communication, making GEMV vulnerable to long-range communication latency on a mesh NoC (L) and requiring adherence to local memory and routing constraints (M and R); and (iii) Sequential GEMV operations introduce costly matrix transpose on a NoC, risking violation of the L property.

Designing fine-grained replication to enable parallelism at minimal communication cost. When tensor dimensions are insufficient to achieve the high parallelism required for decode, we propose fine-grained replication of tensors in LLMs,

[2] L: sequence dimension; L: memory access latency in PLMR.

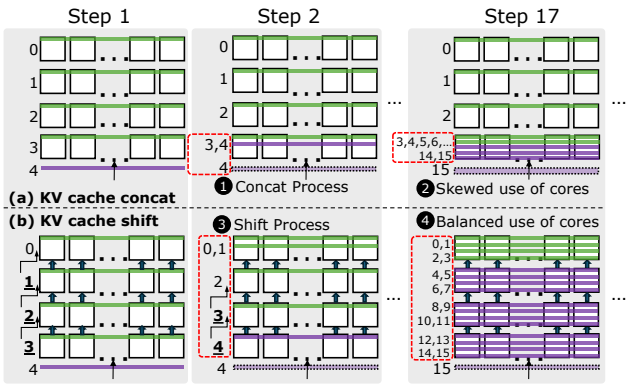


Figure 5: KV cache concatenation vs. KV cache shift

specifically replicating the sequence dimension, where the sequence length equals the prompt length during prefill phase and equals 1 during the decode phase. This approach offers two key advantages: (i) it improves parallelism and ensures balanced loads across all cores, and (ii) it avoids additional communication operations such as allreduce across all cores. As shown by ❶ in Figure 4, the E dimension is partitioned along the y-axis, and the L dimension is replicated along the x-axis, represented as BE_yL^x. Weight matrices W are partitioned across both dimensions, consistent with the prefill phase.

Our fine-grained replication differs from recent work on long-context/sequence inference systems [49, 55], which selectively replicate certain dimensions during the prefill phase rather than the decode phase.

Designing PLMR-compliant distributed GEMV. We found that existing GEMV implementations fail to fully comply with PLMR requirements due to long-range communication and excessive routing resource consumption at each core. To address this, we propose a PLMR-compliant distributed GEMV, utilizing this new implementation throughout the decode phase (as detailed in ❷ of Figure 4). A comprehensive description of this GEMV design is provided in Section 6.

Pre-optimizing model weight placement to avoid matrix transpose. To avoid matrix transpose during decode, we pre-optimize the model weight layout for decode, particularly for the distributed GEMV operation, to eliminate matrix transpose. While this introduces re-placement overhead between prefill and decode phases, the overhead is far smaller than that of sequential matrix transpose during token generation.

Figure 4 illustrates this proposal, detailed in ❸. Specifically, we optimize the placement of weights such as W_O and W_{out} for distributed GEMV in decode, differing from their layout in the prefill phase. This approach also removes transpose operations in calculating $Q@K^T$ during decode self-attention.

4.3 Shift-based KV cache management

KV cache management on PLMR devices is challenging as it requires storing large data across distributed cores while adhering to local memory constraints (M) and distributing KV cache computations to achieve high parallelism (P). To address these, we have the following insights:

Existing concatenate-based management causes skewed core utilization. Current KV cache management methods primarily concatenate the most recently generated KV vectors to the existing cache. Though efficient in shared memory architectures, this concatenate operation leads to highly skewed core utilization on PLMR devices, as shown in ❶ of Figure 5, where only the core in a row is responsible for storing and computing over the generated KV vector. After several token generation steps, this only core quickly becomes the bottleneck, as depicted in ❷ of Figure 5, causing skewed memory usage and violating the M in PLMR. Moreover, the imbalanced KV cache distribution across cores results in inefficient parallelism, violating the P property.

Proposing shift-based management for balanced core utilization. We propose a shift-based KV cache management strategy that evenly distributes cache data across all cores. Instead of concatenating new KV cache vectors at the end, this method performs a balancing shift operation, where each row transfers the oldest KV cache data to the row above, as shown in ❸ of Figure 5. When new KV data arrives, each core checks its local capacity against its neighbors. If equal, upward shifts are triggered, with each row receiving data from below and passing some to the row above. As illustrated in ❹, this ensures even KV cache distribution across all cores.

The upward shifts utilize NoC links in parallel, maintaining high performance and satisfying the P property. The physical placement of KV cache aligns with logical continuity and only involves data movement between the adjacent cores, adhering to the L property. This method also fully resolves the M violation issue observed in the last row of cores with the concatenate-based approach.

4.4 Implementation details

We outline several implementation details below:

Prefill and decode transition. Prefill and decode require distinct strategies. To handle the transition efficiently, we reshuffle KV cache and weights through the fast NoC, which often provides 100s Pbits/s aggregated bandwidth, completing instantly without relying on slower off-chip memory.

Parallelism configuration. WaferLLM uses offline autotuning to select core counts for each model, optimizing latency based on model size, input/output length, memory per core, and prefill/decode phases. It chooses different core counts for each phase, with fast dynamic remapping enabled by high NoC bandwidth. For models with variable input/output lengths, average values are used to maintain near-peak performance. Autotuning runs separately per model to adapt to specific needs.

Variations of self-attention. WaferLLM supports variations of Self-Attention, including Grouped Query Attention [3], Multihead Attention [43], and Multi-query Attention [5].

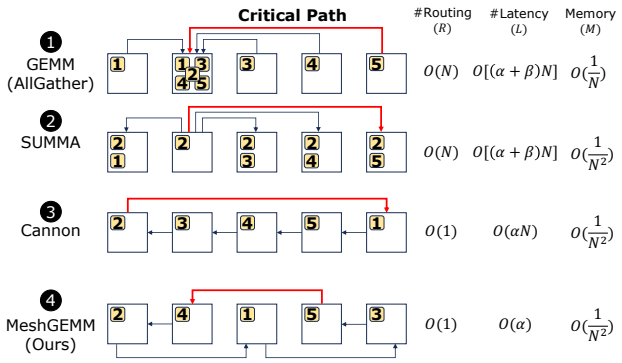


Figure 6: PLMR compliance in distributed GEMM

These differ by performing dist-GEMM, dist-GEMV and dist-GEMM-T locally after grouping by head dimensions.

5 Wafer-Scale GEMM

In this section, we introduce MeshGEMM, a scalable distributed GEMM for massive-scale, mesh architectures.

5.1 PLMR compliance in distributed GEMM

To identify a scalable distributed GEMM for PLMR devices, we define the following metrics: (i) *Routing paths per core*: The number of routing paths per core, with fewer paths ensuring compliance with the R property. (ii) *Latency of critical path*: Maximal latency among all communication paths in each step to transmit submatrix (as the red lines in Figure 6), with less latency adhering to the L property. (iii) *Memory per core*: The memory required per core, with lower usage ensuring the M property.

We analyze current distributed GEMM methods and show how MeshGEMM meets these metrics:

(1) **GEMM via Allgather** is commonly used in GPU and TPU pods for distributed GEMM [32, 35, 56]. Its critical communication path in each step is one core gathering data from the farthest cores, shown as the red line in Figure 6 ❶, and N steps to complete the allgather. Each core creates N routing paths to neighbors in its row and column (violating R). The constraint of R necessitates step-by-step submatrix transmission via intermediate cores, introducing a communication latency of $O[(\alpha+\beta)N]$ along the critical path (violating L), and each core uses $O(1/N)$ memory due to inflated working buffers (violating M).

(2) **SUMMA** is Cerebras' default choice for distributed GEMM on its wafer-scale engine [7]. Its critical communication path in each step is where one core broadcasts data to the farthest core along the column or row, shown by the red line in ❷ of Figure 6. Same to the allgather communication in ❶, each core creates N routing paths (violating R) and spans the critical path with $O[(\alpha+\beta)N]$ latency (violating L) in each step. While SUMMA improves memory usage compared to allgather, requiring only a working set equal to the size of locally partitioned submatrices, it still doubles the peak memory usage.

(3) **Cannon** is mesh-optimized choice for distributed GEMM [6], popular in supercomputers and distributed cluster. Its critical communication path in each step is the head cores send data to the tail cores shown in ❸ of Figure 6. Each intermediate core communicates with two neighbours in a 2D torus and passes through the submatrix from head to tail, which only needs $O(1)$ communication paths and optimal memory usage of $O(1/N^2)$.
Notably, Cannon involves only a constant number of routing paths per core, allowing static routing rules to be assigned for both neighbor communication and the critical path. At each step, submatrices can pass directly from the head to the tail core through intermediate cores, unlike GEMM via allgather or SUMMA, which require step-by-step message transmission. As a result, the critical path incurs only per-hop latency α, without additional per-routing overhead β. However, since the critical path spans N hops, it still incurs $O(\alpha N)$ latency, shown as the red line in ❸, violating the L.

(4) **MeshGEMM (Ours)** is a distributed GEMM which complies with the PLMR model. The critical communication path in each step, which we named as two-hop transmission, is shown as the red line in ❹ of Figure 6, which is significantly shorter than the others. Each core communicates with two two-hop away neighbors(proven in later sections to be scalable for larger-sized mesh architectures), and passes through a one-hop neighbor's communication. This design achieves $O(1)$ communication paths per core needed and optimal memory usage of $O(1/N^2)$, similar to Cannon. Crucially, it bounds the critical path to constantly 2 hops with $O(\alpha)$ complexity, making it uniquely capable of addressing the L property.

5.2 Design intuitions and scalability analysis

Our design involves two phases: (i) We ensure algorithm correctness using a cyclic shifting process for GEMM, and (ii) then use interleaving to bound the critical path latency to a constant. Based on the cyclic shifting and interleaving, we get the **two-hop transmission** communication path, and we prove that it, on this cycle, is the minimal distance required to satisfy the L property.

Cyclic shifting. Cyclic shifting enables MeshGEMM to satisfy the M and R properties by limiting communication to two neighbors and minimizing memory usage. It ensures correct GEMM results, following reasoning similar to Cannon [6]. As illustrated in ❸ of Figure 6, a logical circle of 5 cores is flattened into the physical communication mapping, with a critical path from head core to tail core.

Interleaving. For the flattened communication plan, we would like to minimize the length of the critical path further, thus satisfying the L property. Our key intuition here is

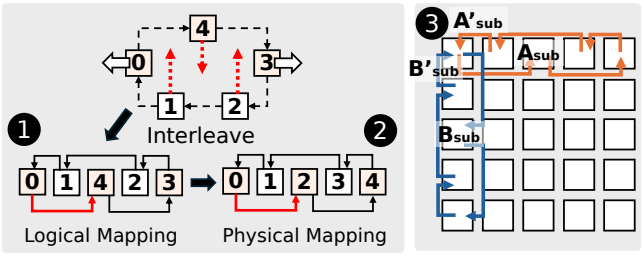


Figure 7: Design intuitions and scalability analysis.

Algorithm 1: INTERLEAVE

Input: index, N
Output: send_index, recv_index

```
if index mod 2 == 0 then
    recv_index = Max (index - 2, 0);
    send_index = Min (index + 2, N - 1);
else
    recv_index = Min (index + 2, N - 1);
    send_index = Max (index - 2, 0);
if index == 0 then recv_index = 1;
if index == N - 1 then
    if N mod 2 == 0 then recv_index = N - 2;
    else send_index = N - 2;
Return send_index, recv_index;
```

to introduce an INTERLEAVE operation to find the mapping relationship from logical to physical, defined in Algorithm 1. As shown by ❶ of Figure 7, MeshGEMM first insert core 1 in between core 0 and 4 and core 2 in between core 4 and 3 to form a logical mapping, and then call the INTERLEAVE operation to get the send to and receive from neighbours' index, resulting in a permutated, equivalent communication plan as shown by ❷ in Figure 7. For example, there are 5 cores total (N=5), so physical core 2 (index=2) sends data to physical core 4 (send_index=4) and receives from physical core 0 (recv_index=0). Figures 6 and 7 illustrate the case with 5 cores as an example, while Algorithm 1 demonstrates that the two-hop communication pattern in MeshGEMM generalizes to mesh architectures of arbitrary size with $N \geq 3$.

Scalability analysis. We can prove that the two-hop distance created by INTERLEAVE cannot be further reduced. The proof relies on the fundamental properties of sequential arrangements: if we attempt to create a circular sequence where each number differs from its neighbors by exactly one hop, we encounter a mathematical impossibility. This can be understood by visualizing the numbers as points on a line - while adjacent numbers can be connected, the endpoints of the sequence cannot simultaneously maintain single-hop differences with their neighbors while forming a circle.

Note that our discussion, based on a 1D array, naturally extends to a 2D mesh, as the 1D array corresponds to the mesh's X-axis and Y-axis due to their symmetry.

5.3 The MeshGEMM algorithm

We outline the key steps of MeshGEMM below:

(1) **Initialization:** Consider $C = A \times B$. MeshGEMM will partition A and B into tiles A_{sub} and B_{sub} along two dimensions, forming $N \times N$ tiles, which are distributed across the cores. Each core receives one tile of A_{sub} and one of B_{sub}. MeshGEMM will then use INTERLEAVE to initialize the neighbors' positions for each core.

(2) **Alignment:** Each core will then align with neighbors to align the input submatrices in a way that ensures every core in the distributed system begins with the appropriate operands for the matrix multiplication process.

(3) **Compute-shift loop:** Each core operates with a compute-shift loop involving N steps of communication and computation. In each step, every core computes the partial sum of its corresponding $C_{sub} = A_{sub} \times B_{sub} + C_{sub}$. Meanwhile, shift A_{sub} along the X-axis and B_{sub} along the Y-axis to get new A'_{sub} and B'_{sub} for the next step computation as ❸ we shown in Figure 7. After N steps, the accumulated C_{sub} is returned.

5.4 Implementation details

Handling non-square mesh. For a non-square mesh $N_h \times N_w$ ($N_h \neq N_w$), the A and B matrices can be logically partitioned into $N_{lcm} \times N_{lcm}$ cores, where N_{lcm} is the least common multiple of N_h and N_w.

Transposed distributed GEMM. The above algorithm can be applied to the computation of $C = A \times B^T$, the dist-GEMM-T in Figure 3 to avoid transposing B on mesh. It does not require alignment before computation and only necessitates N steps of two-hop compute-shift for the right matrix B along the Y-axis. After each shift step, each core computes $C_{\text{sub}} = A_{\text{sub}} \times B_{\text{sub}}$, followed by a ReduceAdd of C_{sub} along the X-axis. After N steps, the final matrix C is obtained.

6 Wafer-Scale GEMV

In this section, we describe MeshGEMV, a scalable GEMV algorithm for PLMR devices.

6.1 PLMR compliance in distributed GEMV

The completion time of a distributed GEMV is primarily determined by an element-wise computation to generate a partial sum result at each core and then an allreduce operation that aggregates partial results from all selected cores and then sends the aggregated results back to all cores for the continuous GEMV. As the analysis in GEMM, in wafer-scale GEMV, we also define the following metrics: (i) *Routing paths per core*: The number of routing paths per core, with fewer paths ensuring compliance with the R property (ii) *Latency of critical*

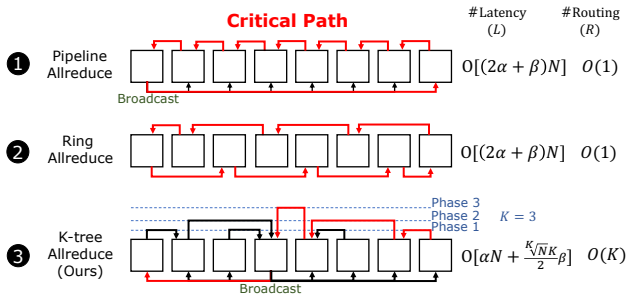


Figure 8: PLMR compliance in distributed GEMV

path: Maximal latency of the whole allreduce communication, mainly determined by the time from when the partial sum is sent from the farthest core to when the aggregated sum is received by the farthest core.

We analyze current distributed GEMV methods and show how MeshGEMV meets these metrics:

(1) **GEMV with pipeline allreduce** is commonly used in TPU pod systems [35] and as the default in Cerebras demo [9]. As illustrated by the red line in the Figure 8 ❶, the critical path of the pipelined all-reduce starts from the farthest core, where partial sums are reduced step-by-step toward the root core, and the aggregated result is then broadcast from root to all cores. While this approach ensures that the routing path per core remains within the R constraint, $O(1)$ per-core routing resource usage, it incurs a total of $2N$ hops and N routing stages, violating the L property.

(2) **GEMV with ring allreduce** is commonly used in GPU pod systems, where it is the default configuration, especially for a large amount of data [27]. As shown by ❷ in Figure 8, the critical path of ring allreduce involves each partial sum traversing all cores in the ring. Similar to pipelined allreduce, this approach maintains $O(1)$ routing paths per core, but incurs a communication latency of $O[(2\alpha+\beta)N]$, thus violating the latency constraint L defined in PLMR.

(3) **GEMV with K-tree allreduce (Ours)**. As analyzed above for pipelined and ring allreduce, the sequential execution of reduce-add operations leads to N routing stages along the critical path. In contrast, K-tree allreduce organizes the reduce-add path as a balanced K-tree, enabling K phases grouped parallel reductions with $O(\sqrt[K]{N})$ cores per group and reducing the critical path to only $\frac{K\sqrt[K]{N}K}{2}$ times routing and N hops. However, this comes at the cost of requiring $O(K)$ routing paths per core.

6.2 The MeshGEMV algorithm

We will outline the key steps of MeshGEMM below:

(1) **Initialization:** Consider $C = A \times B$ and A is a vector. MeshGEMV will partition B into tiles B_{sub} along two dimensions, forming $N \times N$ tiles and distributed across the cores. For A, MeshGEMV will partition it along the vector length, forming N tiles distributed on one axis and replica A on another axis. Each core receives one tile of A_{sub} and one of B_{sub}. Then we determine which cores form a group to obtain aggregated results in each phase based on the K-tree.

(2) **Parallel computation:** In this stage, each core performs a local GEMV $A_{\text{sub}} \times B_{\text{sub}}$ to obtain C_{sub} partial sum.

(3) **Aggregation:** The aggregation step primarily involves using the K-tree allreduce we design. The key steps are as follows: (i) In the 1st-phase, each group performs group reduction and obtains the partial sum of C_{sub} at the root core of each group. (ii) In the kth-phase, the results from the $(k-1)$ th-phase are reduced to the root cores of each group in the kth-phase. After K times repeating, C can be obtained by concatenating the C_{sub} from all K-tree root cores. (iii) Optionally, a broadcast operation from the root core of the K-tree may follow, depending on whether continuous GEMV is required.

Scalability Analysis. As shown in ❸ of Figure 8, this method scales efficiently with parallelism and meets the L property by selecting an appropriate K. It requires $K+1$ paths at the tree root core but allows flexible adjustment of K to address R based on hardware limitations. Compared to pipeline and ring all-reduce, K-tree all-reduce essentially builds pass-through paths between distant cores by consuming routing resources, thereby reducing routing latency.

However, a larger K is not always better, as it depends on N and R constraints. Additionally, larger K increases routing complexity and overhead. Considering these factors, we have chosen $K = 2$ for our current implementation, evaluated in the following sections.

7 Evaluation

We extensively evaluated WaferLLM against various state-of-the-art methods and systems. Our results show that:

(1) WaferLLM achieves orders of magnitude speedup over T10 and Ladder in LLM inference (§7.1);

(2) WaferLLM's MeshGEMM and MeshGEMV perform and scale strongly over the state-of-the-art (§7.2);

(3) WaferLLM's shift-based KV cache management enables over 360-385× more token capacity (§7.4);

(4) WaferLLM on Cerebras WSE-2 achieves 10-20× e2e speedup compared to the optimal performance of SGLang on A100 GPUs-cluster, and about 30-40× speedup compared to a single A100 GPU. WaferLLM also achieves approximately 2.5× better energy efficiency (§7.1,§7.5).

Experiment setup. We evaluate WaferLLM on a server with Cerebras WSE-2. WSE-2 has 850,000 Cores, each with a Compute Engine (CE) operating at a maximum of 1.1 GHz. Each clock cycle can fetch two 32-bit operands from SRAM, perform a multiply-accumulate operation, and then write back

to SRAM. Each core also has a fabric router that can send or receive 32-bit messages from neighbouring cores with a single clock cycle. Additionally, each core contains 48KB of SRAM, with the chip totalling 40GB of aggregated SRAM [24].

We compare WaferLLM with two DNN compilers: (i)T10 [25], the state-of-the-art compiler for AI accelerators with inter-core connections and distributed on-chip memory, and (ii)Ladder [45], the state-of-the-art compiler for shared memory architectures. For T10, we implemented it on WSE-2, treating each core as part of a distributed memory system interconnected by a crossbar, despite the actual mesh topology. T10 maps data to core IDs and fetches data from local SRAM as required. For Ladder, we treated the distributed memory architecture of the chip, interconnected by mesh, as unified memory, requiring collective communication over the NoC to access data.

We use the Nvidia A100 for GPU comparison experiments, which shares the same 7nm process as the WSE-2 for fairness. The experiments utilize up to 16 GPUs across two nodes (2×8 configuration). Within a single node, the eight A100 GPUs are interconnected via NVLink, while inter-node communication is handled through a high-performance InfiniBand (IB) network. We use SGLang [53], one of the highest-performing LLM inference systems for GPU-based experiments.

Experiment metric. To evaluate the maximum per-request throughput performance of LLM inference systems on different hardware, we define Throughput per Request (TPR) as a key metric. TPR is derived from the more widely used Time per Output Token (TPOT), with TPR $= \frac{1}{\text{TPOT}}$.

LLM models. Our evaluation includes various representative LLMs of different sizes and architectures. Specifically, LLaMA3-8B and LLaMA2-13B are widely used open-source LLMs, with LLaMA3 using group-query attention instead of multi-head attention to reduce KV cache usage. For LLaMA2-13B, we modified the model to remove the 4K context length limitation to evaluate system performance across different input and output lengths. Additionally, due to the tensor parallelism constraints, the number of attention heads of a model must be divisible by the number of GPUs used. We did not conduct the multi-GPU experiment of LLaMA-2 on a 16-GPU cluster. CodeLLaMA-34B is a specialized LLM for coding tasks, while QWen2-72B, another popular LLM, is renowned for its high model quality.

7.1 LLM inference

We first report the end-to-end performance of WaferLLM compared to T10 and Ladder on WSE-2 and SGLang on the A100 GPU cluster. Then we further analyze the performance to provide deeper insights by breaking down the execution into prefill and decode phases.

End-to-end throughput. Table 2 shows the inference TPR

[3] No 2×8 GPUs, due to model architecture and tensor parallelism.

Model	Device		Input/Output Seqence Length			
			2048/128	4096/128	2048/2048	4096/4096
LLaMA3-8B	WSE-2	WaferLLM	764.4	604.4	2370.3	2459.0
		T10	4.6	4.5	58.3	94.6
		Ladder	1.2	1.1	7.4	8.7
	A100 GPU (SGLang)	1	34.8	31.1	36.5	78.4
		8	117.2	109.0	128.4	256.1
		2×8	73.7	70.2	79.3	162.5
LLaMA2-13B	WSE-2	WaferLLM	473.9	414	1690.3	1826.0
		T10	2.6	2.5	35.0	58.3
		Ladder	0.7	0.7	4.9	6.1
	A100 GPU (SGLang)[3]	1	20.4	17.1	21.1	47.9
		8	79.6	70.5	86.9	172.4

Table 2: End-to-end LLM inference TPR

of LLaMA3-8B and LLaMA2-13B on WSE-2 and A100 with different input and output sequence lengths. Here, the end-to-end throughput is calculated as the total number of tokens generated during the decode phase divided by the total time spent in the prefill and decode phases. WaferLLM uses core configurations optimized for the best performance with each model. In LLaMA3-8B, we use 660×660 cores for prefill and 360×360 for decode. In LLaMA2-13B, we use 750×750 cores for prefill and 375×375 for decode. CodeLLaMA-34B and QWen2-72B are not included due to the memory constraint of a single WSE-2 chip.

Compared to T10, WaferLLM achieves 160× speedup on average, up to 180×, for short sequence generation tasks such as 4096 and 2048 input context lengths with 128 tokens output. For longer tasks, with input context lengths of 4096 and 2048 tokens and output lengths of 4096 and 2048 tokens, WaferLLM achieves 36× on average and up to 48×. Although T10 designs the compute-shift model that considers the memory constraints (M) and routing resource limits (R) of a PLMR device, it does not account for the cores interconnected by a mesh NoC. Thus, failing to address varying hop distances (L) and scale to millions of cores (P), highlighting the need for new system designs in massive-scale NUMA architectures.

Compared to Ladder, WaferLLM achieves 625× speedup on average, up 677×, for short sequence generation tasks such as 4096 and 2048 input context lengths with 128 tokens output. For longer sequence generation tasks, with input context lengths of 4096 and 2048 tokens and output lengths of 4096 and 2048 tokens, WaferLLM achieves 312× on average and up to 342×. That is because Ladder is designed for shared memory architecture and does not consider the characteristics of the PLMR device, resulting in failure in partitioning LLMs across millions of cores (P), incurring costly long-range NoC communication (L), failure in handling local memory constraints (M) and limited routing resources (R).

Prefill throughput. Table 3 shows the prefill performance for an input sequence length of 4096, using core configurations from 480×480 to 720×720. For CodeLLaMA-34B and QWen2-72B, which exceed the memory capacity of WSE-2 and a single A100, we evaluate a subset of layers and scale the results proportionally due to their uniform layer structure.

WaferLLM achieves significant speedups over T10 and

Model	Methods	WSE-2 Cores			# A100 GPUs (SGLang)		
		480×480	600×600	720×720	1	8	2×8
LLaMA3-8B	WaferLLM	20320.6	25037.2	27686.5	13988.3	17361.6	13994.2
	T10	175.0	156.6	132.8			
	Ladder	61.8	42.3	31.3			
LLaMA2-13B	WaferLLM	13685.1	16854.2	17498.3	7805.1	12287.1	/[3]
	T10	121.3	100.6	81.3			
	Ladder	47.3	33.1	24.2			
CodeLLaMA-34B	WaferLLM	5471.4	7540.1	8526	5382.5	7155.5	6409.2
	T10	49.1	46.8	41.2			
	Ladder	30.1	23.1	17.7			
QWen2-72B	WaferLLM	2785.2	3775.5	4421.6	1677.3	3803.8	3750.5
	T10	24.9	23.5	21.5			
	Ladder	16.8	12.8	10.1			

Table 3: Prefill Throughput per Request (TPR)

Model	Methods	WSE-2 Cores			# A100 GPUs (SGLang)		
		420×420	540×540	660×660	1	8	2×8
LLaMA3-8B	WaferLLM	2699.9	2501.5	2243.3	78.9	260.4	164.6
	T10	418.3	339.4	265.1			
	Ladder	14.6	13.1	11.4			
LLaMA2-13B	WaferLLM	2039.2	1899.4	1739.8	48.7	175.8	/[3]
	T10	341.8	270.8	233.7			
	Ladder	11.0	9.9	9.0			
CodeLLaMA-34B	WaferLLM	1450.8	1407.7	1359.2	26.1	100.4	84.5
	T10	278.2	222.4	193.1			
	Ladder	6.1	6.2	5.8			
QWen2-72B	WaferLLM	839.7	824.3	787.1	10.6	51.2	48.7
	T10	168.5	133.0	114.6			
	Ladder	3.2	3.3	3.4			

Table 4: Decode Throughput per Request (TPR)

Ladder by effectively addressing all PLMR properties, an average speedup of 160× (up to 178×) over T10 and 270-450× over Ladder. As discussed in Section §2, GEMM is the primary bottleneck, and MeshGEMM substantially enhances WaferLLM's prefill performance, analyzed in detail in Section §7.2.

Additionally, WaferLLM scales throughput with increasing cores across all models. For instance, WaferLLM achieves a 1.6× scaleup on QWen2-72B and a 1.4× scaleup on LLaMA3-8B when scaling from 480×480 to 720×720 cores. In contrast, T10 and Ladder fail to scale effectively, with throughput even declining as more cores are added. This is mainly because T10 and Ladder do not account for the spatial locality characteristic (L) of highly uniformly distributed memory, which can lead to frequent long-distance data reads across cores and increased communication overhead.

Decode throughput. Table 4 shows decode throughput for core configurations from 420×420 to 660×660. For CodeLLaMA-34B and QWen2-72B decode benchmark on WSE-2 and single A100, we evaluate a subset of layers and scale the results.

By addressing all PLMR properties, WaferLLM achieves an average speedup of 5.7× (up to 6.5×) over T10 and 217× (up to 260×) over Ladder.

Compared to the 160× speedup WaferLLM achieves over T10 during prefill, it offers only about a 6.5× speedup during decode. However, WaferLLM maintains a consistent 200-500× advantage over Ladder in both prefill and decode stages. This is mainly because prefill involves dist-GEMM computations, which require each data element to traverse cores along the same row or column sequentially. In contrast, the dist-GEMV allreduce operation in decode only requires traversal across cores in the same row or column, without enforcing a

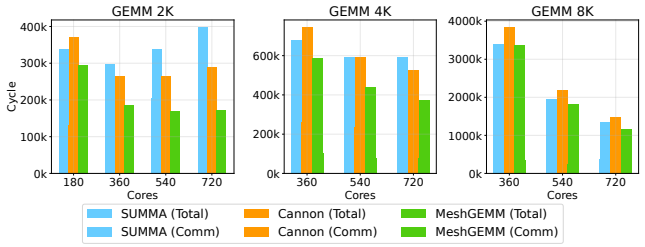


Figure 9: MeshGEMM vs. SUMMA & Cannon

strict access order. While T10's compute-shift paradigm does not consider the spatial locality of cores on a mesh NoC, it still accounts for distributed memory, which allows performance gains when memory (communication) accesses are order-independent. In contrast, Ladder is entirely designed for a shared memory architecture, leading to severe communication overhead regardless of whether memory access order is required.

Comparison with SGLang. As shown in Table 2,3 and 4, WaferLLM demonstrates clear advantages over SGLang + A100 GPU clusters in prefill, decode tasks, and also the end-to-end performance across models ranging from 8B to 72B. More detailed comparisons with GPUs are discussed in Section §7.5.

It is worth noting that T10 and Ladder fail to leverage the powerful hardware capabilities of the WSE-2 when compared to SGLang. For example, in prefill tasks and for Ladder in decode tasks, their performance is even worse than that of SGLang running on a single A100. This highlights that designing high-performance systems and algorithms on wafer-scale chips must carefully consider the PLMR model; otherwise, not only will the massive compute potential of the chip go underutilized, but performance may even degrade due to the highly uniform distributed memory resulting from the large-scale mesh NoC.

7.2 MeshGEMM

We compare MeshGEMM with Cannon [6] and SUMMA [42] on WSE-2 across different core scales and matrix sizes.

Scaling core count. Figure 9 shows that across matrices of different sizes, MeshGEMM can achieve the lowest latency than SUMMA and Cannon by scaling up the number of cores. Compared to the pipeline broadcast in SUMMA and head-to-tail transmission in Cannon, interleave transmission in MeshGEMM can minimize per-step communication overhead and, as much as possible, overlap communication with computation during large-scale fine-grained parallel execution. It demonstrates stronger scalability, maintaining over 70% computational efficiency even near the hardware limit. In contrast, SUMMA and Cannon exhibit poor scalability, with computational efficiency falling below 50% with 720×720 cores, primarily due to the communication overhead in SUMMA and Cannon, increasing with the parallelism scale.

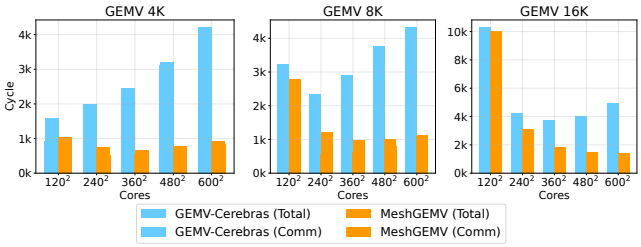


Figure 10: MeshGEMV vs. GEMV-Cerebras

Additionally, increasing the number of cores does not always bring performance gains, especially for small matrix GEMM tasks. For example, in GEMM 2K, when scaling from 360×360 to 720×720 cores, the end-to-end latency of SUMMA and Cannon increases instead of decreasing. This is because the per-core computation cost drops sublinearly as cores increase beyond a threshold (due to fixed overheads like function calls and logic checks). At the same time, communication overhead grows linearly, as seen from the shaded areas in Figure 9, leading to worse overall performance. In contrast, MeshGEMM only shows a slight communication overhead increase at 720×720 cores and maintains stable end-to-end latency. This is because its interleave communication bounds the per-step communication cost to a constant, independent of core count (the total communication overhead grows only because the number of steps increases in all dist-GEMM algorithms). Thus, MeshGEMM can achieve a better overlapping between communication and computation even under large-scale fine-grained parallelism (each core is only responsible for a small amount of data).

Scaling matrix size. We also evaluate MeshGEMM with larger matrix sizes, transforming GEMM into a more computationally intensive operation. At large scales, though the cost of communication becomes less significant, MeshGEMM maintains its scalability and outperforms SUMMA and Cannon by a wide margin, reducing total cycles by around 17%.

An interesting observation in Figure 9 is that for GEMM 8K, communication cycles decrease as core count increases. This occurs because when processing large data volumes, communication is bandwidth-bound rather than latency-bound. Increasing the number of cores not only boosts aggregate compute power to reduce computation overhead but also increases aggregate bandwidth to lower overall communication cost.

7.3 MeshGEMV

We evaluate MeshGEMV and the default GEMV implementation on Cerebras (pipeline allreduce) across various core scales and matrix shapes.

Communication latency bottleneck. Figure 10 shows that communication becomes the primary bottleneck for dist-GEMV computations on WSE-2, especially when the parallelism scale is large relative to the computation per core, where communication overhead can dominate 90% of the total. However, MeshGEMV significantly reduces communication overhead compared to the Cerebras baseline GEMV, achieving about 4.6× higher end-to-end performance. This is because large-scale dist-GEMV requires long-distance allreduce communication, and MeshGEMV uses K-tree Allreduce to maximize parallelism in the allreduce process, minimizing communication and computation costs along the critical path.

Model	LLaMA3-8B	LLaMA2-13B
Concat-based (PagedAttention)	382	16
Shift-based (WaferLLM)	137548	6168

Table 5: Maximum decode output length

GEMV	[1,16K]×[16K,16K]			[1,32K]×[32K,32K]		
TP in SGLang(A100)	1 GPU	8 GPUs	2×8 GPUs	1 GPU	8 GPUs	2×8 GPUs
Time (ms)	0.336	0.253	0.340	1.231	0.341	0.339
MeshGEMV (WSE-2) Time (ms)	0.0012			0.00203		
A100/WSE-2 Energy Ratio	**7.47**	**44.97**	**120.88**	**16.17**	**35.83**	**71.25**

Table 6: Comparing MeshGEMV (WSE-2) with TP in SGLang(A100) of GEMV latency and energy.

Scaling core count and matrix size. We also evaluate MeshGEMV with larger matrix sizes. For GEMV 8K and 16K compared to 2K, the baseline shows a trend that the end-to-end overhead first decreases and then increases. The initial decrease comes from increased core count, where the higher aggregate compute power and bandwidth reduce computation and communication costs. However, as the core count grows, the communication latency of allreduce eventually overtakes computation and bandwidth as the dominant bottleneck. In contrast, MeshGEMV's inflection point appears later, which means that for the same data size, MeshGEMV's communication overhead grows much more slowly with increasing core count compared to the baseline, demonstrating better scalability. This enables more flexible on-chip mapping policies, such as using more cores to store additional LLM parameters to meet memory limitations (M) without introducing significant extra overhead.

7.4 Shift-based KV cache management

We also compare the shift-based KV cache management with the concat-based KV cache management in PagedAttention. We evaluate KV cache capacity on LLaMA3-8B and LLaMA2-13B using the same settings as the end-to-end inference evaluation in Section §7.1. Table 5 shows that WaferLLM's shift-based KV cache management supports 360× and 385× more tokens than the concat-based method for LLaMA3-8B and LLaMA2-13B, respectively. This improvement results from balanced core utilization and the resolution of skewed data issues achieved by the shift-based approach.

7.5 Comparison with GPUs

We compare WaferLLM against the state-of-the-art LLM inference system on GPUs using Cerebras WSE-2 and NVIDIA

Prefill (4K CTX)	LLaMA3-8B			LLaMA2-13B	
SGLang(A100) TPR	1 GPU	8 GPUs	2×8 GPUs	1 GPU	8 GPUs
	13988	17361	13994	7805	12287
WaferLLM WSE-2 TPR	27686			17498	
A100/WSE-2 Energy Ratio	**0.05**	**0.34**	**0.84**	**0.06**	**0.30**

Table 7: Comparing WaferLLM (WSE-2) with SGLang(A100) in prefill throughput and energy.

Decode (4K CTX)	LLaMA3-8B			LLaMA2-13B	
SGLang(A100) TPR	1 GPU	8 GPUs	2×8 GPUs	1 GPU	8 GPUs
	78	260	164	48	175
WaferLLM WSE-2 TPR	2700			2039	
A100/WSE-2 Energy Ratio	**0.92**	**2.22**	**7.02**	**1.13**	**2.49**

Table 8: Comparing WaferLLM (WSE-2) with SGLang(A100) in decode throughput and energy.

A100. To compare against the H100 fairly, we would need access to the WSE-3, which is a 5nm manufacturer but unavailable to us.

GEMV. We compared MeshGEMV with the GEMV parallelization strategy that follows SGLang's [53] multi-GPU tensor parallelism, while the computation on each GPU is accelerated using the cuBLAS library. Shown by Table 6, compared to a single A100 GPU, MeshGEMV outperforms by 280-606× with different matrix sizes, showcasing the advantages of providing substantial memory bandwidth through wafer-scale devices. This also translates to 7.5-16× greater energy efficiency, reflecting the benefits of wafer-based connections (connecting on-chip memory) over PCB-based ones (connecting off-chip HBM) in GPUs.

Distributed GEMV on multi-GPUs shows limited scalability: a single A100 yields the highest energy efficiency per FLOP, and performance improves only 1.32× from one to eight GPUs, then degrades at sixteen. This inefficiency stems from memory intensity and communication overhead over NVLink and IB. In contrast, MeshGEMV exploits WSE-2's low-latency NoC to scale across hundreds of thousands of cores, achieving 166-210× higher performance and 45-70× better energy efficiency at peak compared to the best GPU cluster results. Moreover, GEMV exhibits similar performance for 16K and 32K tasks on two machines with 16 GPUs, mainly because communication overhead dominates the computation in distributed GEMV across two nodes.

Despite these advantages, MeshGEMV does not achieve the theoretical 7,000× improvement. Profiling identifies three contributing factors: (i) WSE-2 cores, still in their second generation, cannot fully overlap memory access and computation; (ii) edge cores are underutilized; and (iii) NoC long-range communication overhead persists, despite MeshGEMV mitigating it effectively. We anticipate these gaps will continue to narrow as wafer-scale accelerators mature.

LLM inference. The maximum TPR achievable on GPU clusters is significantly lower than on WSE-2. We compare WaferLLM with SGLang on an A100 multi-GPUs cluster and find that WaferLLM delivers a 6-20× higher TPR across a range of input/output lengths and model sizes from 8B to 72B, with the gap widening for longer outputs and larger models (Table 2). As shown in Tables 7 and 8, scaling SGLang from 1 to 8 GPUs yields only 1.2-1.6× prefill and 3.3-3.6× decode speedups, far below ideal linear scaling. Performance further degrades when scaling to 16 GPUs, due to inter-node communication bottlenecks. As a result, peak TPR for dense models is typically reached within a single 8-GPU node, where WaferLLM still outperforms SGLang up to 20×. For use cases demanding high per-request throughput, wafer-scale accelerators are substantially more capable than conventional xPU-based systems.

In terms of energy efficiency, though a WSE-2 chip has 47× the area and 37× the power and cost of an A100, WaferLLM still achieves a 2-2.5× energy efficiency advantage at SGLang's optimal multi-GPU result, owing to GPUs' nonlinear scaling on decode. This advantage is especially valuable for long-output scenarios like test-time scaling, where serving cost is critical. However, compared to GEMV, WaferLLM's decode energy efficiency advantage is reduced due to: (i) limited local SRAM (48KB) on WSE-2 cores, which hinders efficient tensor parallelism and necessitates pipeline parallelism, causing up to 5× underutilization; and (ii) GPU-optimized LLaMA models with narrow layers that constrain layer placement and worsen utilization on WSE-2.

8 Implementation Detail and Future Direction

We discuss the current limitations of WaferLLM and wafer-scale accelerators and envision their future solutions:

Hardware architecture. The performance of WaferLLM is currently constrained by execution bubbles caused by the need for pipeline parallelism. Increasing a core's local memory by 5-6× could eliminate the need for pipeline parallelism, enabling full tensor parallelism, as on vLLM and SGLang. Wafer-scale chip designers are already moving in this direction. Cerebras WSE-3 retains the same NoC configuration but improves per-core efficiency and local memory, while Tesla's Dojo incorporates 1MB of per-core memory.

Memory-to-Compute Ratio. LLM decoding demands a near 1:1 memory-to-compute ratio. However, GPUs like A100 have limited on-chip SRAM, forcing frequent off-chip memory access and yielding a poor ratio of 1:312 (FP16). Multi-GPU setups exacerbate this due to heavy inter-GPU communication. In contrast, Cerebras WSE-2 maps most model weights onto on-chip memory via a low-latency NoC, achieving near-ideal locality and approaching a 1:1 ratio. To fully realize this balance on mesh-based NoCs, adherence to the PLMR model is essential, enabling WaferLLM to outperform GPU-based systems by orders of magnitude in TPR.

Handle reliability issues. Currently, Cerebras WSE-2/3 handles faults by hardware, only exposing healthy cores (organized in a mesh) to software, with no explicit handling required at the software level. Moreover, redundant cores and

links are built in at fabrication, and the on-chip SoC dynamically remaps and reroutes around defects at runtime, ensuring minimal performance impact at a low redundancy cost. Meanwhile, wafer-scale chip makers recently reported a 93% functional wafer area, higher than 70–80% in commercial GPUs [8], due to the smaller area per core design. Over two years of WSE-2 deployment, we have observed high reliability, confirming the effectiveness of these fault-tolerance mechanisms in real-world use.

Various model architecture. WaferLLM is also beneficial for MoE as it shares key operators with dense LLMs, including MeshGEMM, MeshGEMV, and shift-based KV cache management. The main difference is the all-to-all communication between attention and expert layers, which we implement using WSE-2's NoC multi-cast operations. Further optimizations for sparse models, such as offloading and sparse attention, are among our future research.

Beyond Cerebras WSE. While evaluated on Cerebras WSE, the PLMR model generalizes to emerging mesh-like architectures such as Tesla Dojo, which also feature hundreds of thousands of cores with local memory and constrained NoC routing. Variants like 2D torus or hybrid mesh-switch topologies also conform to PLMR. Our design for MeshGEMM and MeshGEMV targets worst-case 2D mesh and remains competitive across such platforms. Beyond on-chip meshes, chip-to-chip mesh interconnects, as seen in Tenstorrent's core- and card-level meshes, also align well with PLMR. Looking ahead, advancements in wafer-scale integration, such as TSMC's projected 40× density increase by 2027, further reinforce the long-term applicability of our approach.

9 Related Work

Deep learning frameworks and compilers. Current deep learning frameworks and compilers, such as PyTorch, TensorFlow, and XLA [1, 10, 28, 34, 36, 37, 45, 51, 54, 57], are designed for shared memory architectures and use a tile-based "load-compute-store" computation model. While effective for shared memory, this model ignores the unique characteristics of PLMR devices, making it inefficient for wafer-scale AI chips. LLM frameworks such as vLLM and TensorRT-LLM [20, 56] have emerged to support modern LLMs but rely on frameworks and compilers designed for shared memory architectures (e.g., PyTorch [34]), inheriting similar limitations on wafer-scale chips.

Distributed GPU and TPU systems. The on-chip distributed memory architecture could theoretically be treated as a distributed LLM system, as studied in prior works [20, 23, 35, 49, 52, 56]. However, such systems, designed for GPU and TPU pods (up to thousands of nodes), rely on more capable routers and lack local memory constraints, making them misaligned with the PLMR model. These approaches are complementary to our focus on on-chip scaling.

Systolic array. Systolic array architectures [22], used in AI accelerators such as Amazon Trainium and Google TPU, focus on the design of small cores rather than larger wafer-scale accelerators. With limited processing elements (usually up to hundreds) in a core, they are not PLMR devices but complement WaferLLM. For example, a Cerebras WSE core could employ a systolic array to accelerate local GEMM operations.

Dataflow architectures. Prior research has explored computation on dataflow architectures that account for inter-core connections. TENET [26] maps computation spatially and temporally to connected cores in a dataflow pattern. DISTAL [50] enables scheduling over distributed clusters using a dataflow approach. SambaNova [14] combines model and pipeline parallelism for DNN execution. However, none of these works scale computation to wafer-scale chips.

Wafer-scale allreduce. Recent research [27] has investigated wafer-scale allreduce, but a single allreduce cannot fully parallelize GEMV or support full LLM inference as achieved by WaferLLM. Additionally, this prior work is a specific instance of the K-tree allreduce proposed in WaferLLM.

10 Conclusion

We envision this paper as a foundational step in exploring the potential of wafer-scale computing for LLMs. The simple yet effective PLMR model has revealed significant opportunities, guiding the development of the first wafer-scale LLM parallelism solution and scalable GEMM and GEMV algorithms for wafer-scale accelerators. Despite the limitations of the current software stack for wafer-scale devices, our approach achieves orders-of-magnitude improvements in both performance and energy efficiency. We hope this work inspires greater focus on wafer-scale computing and advances the path toward a more sustainable future for AI.

11 Acknowledgments

We sincerely thank our shepherd, Marco Canini, and the OSDI reviewers for their insightful feedback that greatly improved this paper. We are also grateful to Yuqing Xia (Microsoft) for her valuable input during the development of this work.

We acknowledge the hardware resources provided by the Edinburgh International Data Facility (EIDF) and the Edinburgh Parallel Computing Centre (EPCC), particularly their support in granting access to the Cerebras WSE systems. We thank Nick Johnson for coordinating access and Mark Parsons for supporting the hardware acquisition and our project.

Finally, we appreciate the generous assistance of Leighton Wilson and Mathias Jacquelin from Cerebras for their patience and detailed responses to our many questions regarding CSL and the Cerebras WSE.

References

[1] M. Abadi, P. Barham, J. Chen, et al. TensorFlow: A system for large-scale machine learning. *OSDI 2016*, 2016.

[2] Advanced Micro Devices. AMD optimizes EPYC memory with NUMA. White paper, Advanced Micro Devices, Inc., 2023.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[4] AMD. AMD XDNA adaptive architecture, 2023. Accessed: 2024-11-29.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] Lynn Elliot Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[7] Cerebras Systems. GEMM with collective operations. Accessed: 2024-10-05.

[8] Cerebras Systems. 100× defect tolerance: How cerebras solved the yield problem, 2022. Accessed: 2025-04-29.

[9] Cerebras Systems. Benchmark GEMV collectives, 2023. Accessed: 2024-11-29.

[10] T. Chen et al. TVM: An automated end-to-end optimization stack for deep learning. *SSP 2018*, 2018.

[11] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. Parallel matrix transpose algorithms on distributed memory concurrent computers. *Parallel Computing*, 21(9):1387–1405, 1995.

[12] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024.

[13] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[14] Mark Harris. SambaNova's new AI chip and the quest for efficiency, 2023. Accessed: 2024-11-29.

[15] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Hanyu Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on GPUs. *arXiv preprint arXiv:2311.01282*, 2023.

[16] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

[17] Norman Jouppi, Cliff Young, et al. Tensor processing units for machine learning: An introduction. Technical report, Google Inc., 2017.

[18] Patrick Kennedy. Tenstorrent Blackhole and Metalium for standalone AI processing, 2024. ServeTheHome, Hot Chips 2024 Coverage.

[19] Jinwoo Kim, Venkata Chaitanya Krishna Chekuri, Nael Mizanur Rahman, Majid Ahadi Dolatsara, Hakki Mert Torun, Madhavan Swaminathan, Saibal Mukhopadhyay, and Sung Kyu Lim. Chiplet/interposer co-design for power delivery network optimization in heterogeneous 2.5-d ICs. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 11(12):2148–2157, 2021.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[21] Mark LaPedus. TSMC bets big on advanced packaging, 2023. Accessed: 2024-11-29.

[22] Ching-Jui Lee and Tsung Tai Yeh. ReSA: Reconfigurable systolic array for multiple tiny DNN tensors. *ACM Transactions on Architecture and Code Optimization*, 21(3):43:1–43:24, 2024.

[23] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[24] Sean Lie. Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning. *IEEE Micro*, 43(3):18–30, 2023.

[25] Yiqi Liu, Yuqi Xue, Yu Cheng, Lingxiao Ma, Ziming Miao, Jilong Xue, and Jian Huang. Scaling deep learning computation over the inter-core connected intelligence processor with T10. In *Proceedings of the ACM*

SIGOPS 30th Symposium on Operating Systems Principles, pages 505–521, 2024.

[26] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. TENET: A framework for modeling tensor dataflow based on relation-centric notation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 720–733, 2021.

[27] Piotr Luczynski, Lukas Gianinazzi, Patrick Iff, Leighton Wilson, Daniele De Sensi, and Torsten Hoefler. Near-optimal wafer-scale reduce. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 334–347. ACM, June 2024.

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

[29] Xiaoning Ma, Qinzhi Xu, Chenghan Wang, He Cao, Jianyun Liu, Daoqing Zhang, and Zhiqiang Li. An electrical-thermal co-simulation model of chiplet heterogeneous integration systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 32(10):1769–1781, 2024.

[30] Meta AI. Introducing MTIA: Meta's next-generation training and inference accelerator for AI, 2024. Accessed: 2024-12-10.

[31] Microsoft Azure. Azure Maia: For the era of AI from silicon to software to systems, 2023. Accessed: 2024-11-29.

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[33] OpenAI. Openai o3 and o4-mini system card. https://openai.com/index/o3-o4-mini-system-card/, 2025. Accessed: 2025-05-03.

[34] A. Paszke, S. Gross, S. Chintala, et al. Automatic differentiation in PyTorch. *NIPS 2017*, 2017.

[35] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[36] J. Rock et al. XLA: Optimizing TensorFlow for high performance. *Google Research*, 2017.

[37] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[38] Shukri J. Souri, Kaustav Banerjee, Amit Mehrotra, and Krishna C. Saraswat. Multiple Si layer ICs: motivation, performance analysis, and design implications. In *Proceedings of the 37th Annual Design Automation Conference*, pages 213–220, 2000.

[39] Cerebras Systems. Cerebras and g42 break ground on condor galaxy 3, an 8 exaflops ai supercomputer. https://cerebras.ai/press-release/cerebras-g42-announce-condor-galaxy-3, 2024. Accessed: 2025-05-03.

[40] Cerebras Systems. Cerebras powers perplexity sonar with industry's fastest ai inference. https://www.cerebras.ai/press-release/cerebras-powers-perplexity-sonar-with\-industrys-fastest-ai-inference, 2025. Accessed: 2025-05-03.

[41] Emil Talpes, Douglas Williams, and Debjit Das Sarma. DOJO: The microarchitecture of Tesla's exa-scale computer. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–28, 2022.

[42] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, pages 5998–6008. Curran Associates, Inc., 2017.

[44] James Wang. Cerebras brings instant inference to mistral le chat. https://cerebras.ai/blog/mistral-le-chat, 2025. Accessed: 2025-05-03.

[45] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan

Yang, Ting Cao, et al. Ladder: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 307–323, 2024.

[46] Tianqi Wang, Fan Feng, Shaolin Xiang, Qi Li, and Jing Xia. Application defined on-chip networks for heterogeneous chiplets: An implementation perspective. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1198–1210, 2022.

[47] Wikipedia contributors. Static random-access memory, 2024. Accessed: 2024-12-10.

[48] Wikipedia contributors. Wafer-scale integration, 2024. Accessed: 2024-12-10.

[49] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. LoongServe: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 640–654. ACM, 2024.

[50] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. DISTAL: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 286–300, 2022.

[51] Y. Zhao et al. Ansor: A compiler stack for auto-tuning tensor programs. *IEEE Transactions on Software Engineering*, 2020.

[52] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[53] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

[54] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

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

[56] Yuxiao Zhou and Kecheng Yang. Exploring TensorRT to improve real-time inference for deep learning. In *2022 IEEE 24th International Conference on High Performance Computing & Communications*, pages 2011–2018. IEEE, 2022.

[57] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

BLITZSCALE: Fast and Live Large Model Autoscaling with $O(1)$ Host Caching

Dingyan Zhang, Haotian Wang†, Yang Liu†, Xingda Wei✉[1], Yizhou Shan[2], and Rong Chen, Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University [2]Huawei Cloud

Abstract

Model autoscaling is the key mechanism to achieve serverless model-as-a-service, but it faces a fundamental trade-off between scaling speed and storage/memory usage to cache parameters, and cannot meet frequent scaling requirements across multiple hosts. The key problem is that data plane performance is slow, and scaled instances remain stopped while parameters are loading.

In this paper, we first show that the data plane can be made *fast* with no or $O(1)$ caching by loading parameters through the compute network between GPUs because: (1) its speed is comparable to host cache and is underutilized, and (2) scaling multiple instances requires no or $O(1)$ caching with network-optimized multicast. Second, autoscaling can be made *live* by breaking the scaling abstraction for inference from a coarse-grained instance-level to a fine-grained layer-level. This allows us to offload the layer computation from the overloaded serving instances to the scaled ones without waiting for the parameters to be fully loaded.

Under real-world workloads, our system BLITZSCALE achieves up to 94 % lower tail latency reductions compared to state-of-the-art autoscaling system (ServerlessLLM), and it reduces the GPU time used for serving by 49 % when compared with serving systems that do not support autoscaling like DistServe and vLLM with the same service-level-agreement.

1 Introduction

Recent years have seen rapid growth in applications powered by deep learning models like large language models (LLMs) [13, 53, 34, 54, 61]. Due to the huge computation requirements, these models are typically served in model-serving-as-a-service systems (MAAS) [63, 11, 23, 76, 20, 10, 29], which manage a cluster of accelerators (e.g., GPUs) and provision an appropriate number of serving *instances* containing GPUs to each model deployed.

An MAAS system has two design objectives: *maximizing goodput*—the number of requests that meet the service level objective (SLO), and *minimizing instances provisioned* to each model to improve hardware utilization. Achieving both is challenging due to the unpredictable short-term fluctuations in a model's instance demands, (5 × required within 2 seconds), because the request arrival rate bursts at seconds-level [45, 80], where the memory usage of each request is also unpredictable due to the auto-regressive nature of LLMs [45, 80, 29] (see also Figure 1 and §2.2).

Model autoscaling is a promising solution [76, 11, 29, 3]. With autoscaling, a MAAS system only provisions the average number of instances required over the long term for each served model, which remains relatively stable [71]. This improves utilization. Upon bursts, the system automatically scales new instances, avoiding SLO violations due to request queueing caused by insufficient instances provisioned.

Autoscaling speed is critical in minimizing SLO violations because the queued requests are not served until the instances are scaled. For instance, the inference time of a Llama3-8B is 80-900 ms on commodity GPU (A800), while users expect a tight response time (< 1 second) for scenarios like chatbot [12, 81, 33]. Meeting such tight SLO requires less than 500 ms scaling time, but achieving this is challenging especially for LLMs with 10-400 GB parameters. The key reason is the **slow data plane** of autoscaling—the process of loading the model parameters to instances' GPUs. While high bandwidth SSDs are utilized in current work [29], the speed provided by SSDs of GPU servers (2-10 Gbps per GPU [35, 9, 27]) is still far from ideal. For instance, loading Llama3-8 B to a GPU takes 12.8 seconds with 10 Gbps SSD. Another factor obstructing fast scaling is that existing scaling methods are **stop-the-world**: the scaled instances cannot serve requests until all parameters are loaded. This implies that autoscaling is directly bottlenecked by the data plane.

To mitigate the above issues, state-of-the-art systems like ServerlessLLM further adopt a multi-tiered caching system by caching model parameters in the host (CPU) DRAM to accelerate the data plane [36, 41]. Under cache hit, they can leverage the fast CPU-GPU link (e.g., 256 Gbps PCIe) to load parameters. However, achieving a high hit rate is unfeasible: ServerlessLLM reports a hit rate of 40–75 %, which is confirmed by us (see §3). The root cause is that a MAAS typically hosts many models, thus achieving a 100 % cache hit requires caching all these models on the DRAM of each host, clearly impractical. Vendors typically host many models because there are hundreds of popular open-source model families designed for different purposes [28]. Meanwhile, each model family has different scales for balancing the serving cost and accuracy [4]. Finally, developers can upload their customized fine-tuned models based on open-source models [10].

To achieve fast model scaling without relying on cache hit, we make the following two key contributions:

1. Data plane made fast with $O(1)$ or no caching with

†These authors contributed equally to this work.

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

compute network multicast. First, a MAAS is backed by fast GPU-GPU/CPU compute fabrics [50], which are 100–400 Gbps RDMA and even 16 Tbps NVLink [35, 9, 27]—much faster than SSD and comparable (or even faster than) CPU-GPU PCIe. The compute fabric is used for data transfer during serving and we found it largely under-utilized, i.e., up to 7.4 % of total bandwidth even in network-heavy workloads like serving LLMs with prefill and decode disaggregation [55, 81, 38, 19, 75] (§3). Thus, we can borrow such fast links for accelerating the data plane of autoscaling.

Second, network-based data plane requires no or minimal caching to achieve fast scaling. Specifically, if a model is already deployed on some instances, we can directly multicast the parameters from deployed instances through the network, eliminating the need for caching. Such multicast is extremely efficient because a serial forwarding multicast [66] can load bulk data (e.g., model parameters), regardless of the number of receivers. Even if no instance is deployed, multicast can be done with $O(1)$ host caching by simply broadcasting the parameters from the host with the cached model. This $O(1)$ caching per-model allows us to avoid all cache misses since the aggregated host memory of all machines is sufficient to cache all models served by a MAAS.

Although fast networking can significantly accelerate the data plane with minimal caching, a stop-the-world loading remains a bottleneck in cases when the networking is not fast enough. For example, to achieve at most 40 % SLO violations when serving a BurstGPT workload with Qwen2.5-72B, the system needs to achieve a tight 500 ms stop time. Achieving so requires 576 Gbps per-GPU[1] parameter transfer bandwidth, far exceeding the available bandwidth of typical compute network setups (e.g., 200 Gbps per-GPU) and even when caching at the host (256 Gbps PCIe). Thus, we argue that an ideal parameter loading should be *live*: before the data loading finishes, the scaled instance should be able to serve requests.

2. Data plane made live with fine-grained scaling abstraction and cooperative execution. Model scaling cannot be live using traditional on-demand data loading techniques commonly found in serverless computing [74, 69, 39] or inference loading overlap in PipeSwitch [15] (§4), because an instance can only emit results once all the parameters are loaded. This stop is rooted in the coarse-grained scaling abstraction of existing systems: they can only scale and serve at the instance level. To realize live scaling, our key insight is that *models can be served in a fine-grained, layer-by-layer manner*. With this fine-grained layer-wise scaling, we can offload part of the layer's computation from overloaded instances to scaled instances with cooperative execution, thus improving the overall serving throughput even before the scaled instance has loaded all the parameters.

Challenges and solutions. First, utilizing network-based multicast is non-trivial in our setup. Though the mechanism of multicast is simple, i.e., simply forwarding parameters between instances with the network, the challenges lie in generating the multicast plan, i.e., determining how the data flows between instances. First, we need to quickly generate an efficient plan online on diverse network topologies since our sources and destinations are dynamically determined, but generating an optimal plan is NP-hard on heterogeneous networks in serving clusters [18]. Second, we need to avoid network interference between the scaling and serving, otherwise, we observed a 1.5 × longer scale time and 50% degraded tail TBT (§4). Current solutions [22, 19, 32] mainly target offline scenarios like training, so they can tolerate long plan generation time and don't need to consider interference from serving workloads. To address the issue, we propose a model-aware multicast planner, which leverages the key features of compute network and the static data flow in model serving to quickly generate a near-optimal, interference-free multicast plan for scaling (§5.1).

Second, it is challenging to schedule how requests are executed between deployed and live scaling instances, i.e., which instance executes which layers. The challenge lies in the fact that the serving capability of the scaling instances is limited—it can only execute layers with parameters loaded, and this capability is dynamically changing. A naive best-effort scaling that executes as many layers as possible cannot balance the load because at the beginning of autoscaling, the new instances can only execute few layers, with requests still queued at the overloaded instances. A better solution is to adjust the load holistically by considering future incoming layers, and we realize this with a ZigZag pipeline scheduling and achieve 50% tail latency reduction under bursty workloads (§5.2).

Demonstration with BLITZSCALE. We built BLITZSCALE, an MAAS system with the fastest autoscaling speed with $O(1)$ caching. We adopted a global parameter pool to cache the model parameters across all the machines, and integrated the aforementioned interference-free multicast plan for scaling and efficient ZigZag scheduling-based live scheduling. To show the effectiveness of BLITZSCALE, we evaluated BLITZSCALE by running real-world traces (i.e., BurstGPT [71], AzureCode and AzureConv [14]) across a variety of recent models with different sizes and architectures, including Llama3-8B, Mistral-24B, and Qwen2.5-72B. First, BLITZSCALE has 47–75 % shorter time-to-first token, and has up to 94 % shorter time-between-tokens than the state-of-the-art work (ServerlessLLM [29]). Second, compared to serving systems without autoscaling support, i.e., vLLM [44] and DistServe [81], BLITZSCALE reduces the GPU used for serving a single model by 49 % with no SLO violations compared to an over-provisioning setup that provisions the GPUs based on the maximum request rate. BLITZSCALE is open-sourced at https://github.com/blitz-serving/blitz-scale.

[1] 72 B model requires at least four GPUs per-instance for serving.

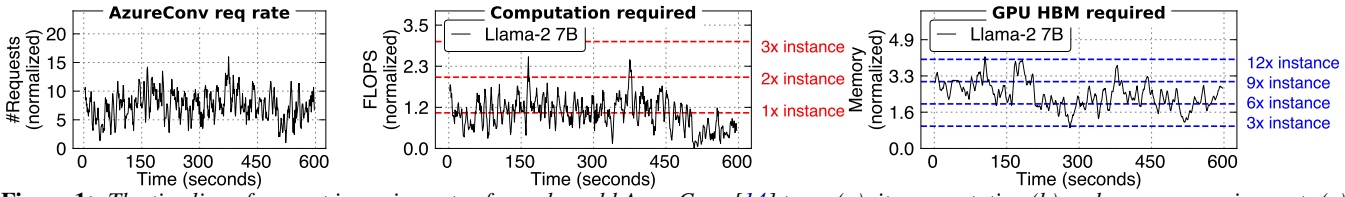


Figure 1: *The timeline of request incoming rate of a real-world AzureConv [14] trace (a), its computation (b) and memory requirements (c) when serving this workload without SLO violation.*

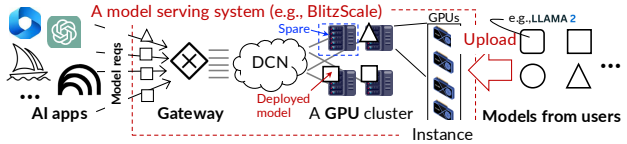


Figure 2: *An illustration of Model as a Service (*MAAS*) system. DCN states for data center network.*

2 Background: MAAS and Autoscaling

2.1 System setup: model-serving-as-a-service (MAAS)

MAAS system setup and the serving instance. BLITZSCALE targets a MAAS scenario [63, 11, 23, 76, 20, 10, 29]: the cloud allows the users to deploy their model serving on its managed hardware, and only charges users based on the number of requests processed within SLO [10, 20]. The model can be popular open-source models like Qwen [4] or customized models uploaded by the users. Thanks to the above pricing strategy, the cloud can dynamically adjust hardware resources to a specific model to maximize its hardware utilization. Figure 2 shows a typical deployment of a MAAS system. For each user-deployed model, the system dynamically allocates the required hardware resources (GPUs) to serve the inference requests of the model. Note that due to the diversity of AI applications, there would be hundreds or even thousands of models served simultaneously by a MAAS system [21].

In this paper, we use *instance* to denote a set of GPUs storing a complete copy of a model parameter for serving this model. An instance can have a single GPU or multiple GPUs when the parameter is large and sharded across them, e.g., with tensor parallelism [47]. Because each instance has a maximal serving throughput, the cloud can deploy multiply instances of the same model by provisioning multiple sets of GPUs, where the number of instances is dynamically scaled based on the incoming request rate. BLITZSCALE supports autoscaling with all existing model serving methods at the instance level.

Serving within an instance: non-LLM & LLM. Each serving instance processes requests in the following workflow: it queries the model in a layer-by-layer computation paradigm (see Figure 9 (a)) and gets the final results. For simple models like vision models, the model is queried once with the input data (image). On the other hand, for large language models (LLMs) [65], the request queries the model multiple times: the model is first queried with input text (prompt) to produce a result (token). This first query is typically termed *prefill*. The token is then used to generate subsequent tokens iteratively until the model returns an end-of-sequence token. The auto-regressive phase is termed *decode*.

We note two important features of LLMs. First, the performance for prefill and decode is measured separately. Prefill is evaluated with the time-to-first-token (TTFT) while decode is evaluated using time-between-tokens (TBT). Second, the LLM query is stateful: the intermediate results—usually termed as *KVCache*—are cached in GPU memory during the auto-regressive phase of a request for acceleration.

Serving across instances: prefill and decode (PD) disaggregated LLM serving. Observing the different computing paradigms of prefill and decode, recent works propose separating the instances for prefill and decode (PD disaggregation) when processing serving requests [55, 81, 38]. Specifically, for each request, one instance processes the prefill phase (prefill instance) and another instance (decode instance) processes the decode phase. This paradigm requires excessive data movement between the two instances because the prefill instance needs to transfer the KVCache to the decode instance. BLITZSCALE works for both PD disaggregated and non-disaggregated LLM serving.

2.2 Dynamic hardware demands when serving a model

Determining the hardware requirements, i.e., the right number of instances for serving a model is challenging because the hardware demands are unpredictable and fluctuating. First, the incoming request rate for a serving workload fluctuates over time and is hard to predict [29, 59, 80]. Figure 1 (a) presents the number of requests sent to a single model service over time from a real-world trace—BurstGPT [71]: the incoming inference requests increase 5 × within 2 seconds with no predictable trend. Since the FLOPS of an instance is fixed, the unpredictable rate causes the computation requirement—FLOPS required to finish the pending requests within SLO—unpredictable. Figure 1 (b) confirms this by measuring the requirement of the prefill instances when serving the BurstGPT with Llama2-7B.

Second, serving modern models like LLM has non-trivial and unpredictable memory requirements. As shown in Figure 1 (c), the KVCache usage of the decode instances is multiple times larger than the memory capacity of a single instance and fluctuates over time (3–12 ×) when serving the BurstGPT workload with Llama2-7B. The root cause is that the KVCache of requests must be stationary in GPU memory during the decode phase. The KVCache of requests are large,

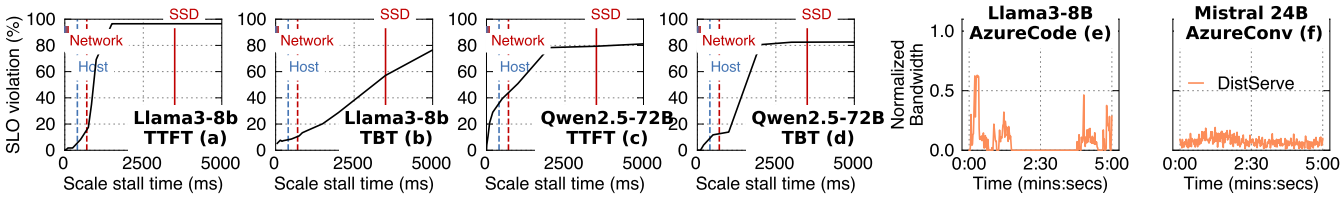


Figure 3: *A characterization of SLO attainment for different inference cases (a)–(d) with varied duration of autoscaling stops on BurstGPT [71]. (e) and (f): an analysis of compute network usage in serving workloads. The evaluation setup is in §6.*

e.g., 190–760 GB for Llama2-7B to serve BurstGPT, and the stay time is unpredictable due to the auto-regressive nature of LLMs. To avoid performance losses when out of memory, a MaaS system must provision sufficient instances to hold KVCache from ongoing requests, so the number of instances required by a model also unpredictably fluctuates.

2.3 Model autoscaling for handling dynamic demands

Model autoscaling, which dynamically deploys[2] serving instances on spare GPUs to scale up the serving capability, is a promising solution to handle fluctuated and unpredictable computation and memory demands [29, 76, 11]. The rationale is that though a single model's workload is unpredictable, the aggregated workloads of all models served by a platform are relatively stable [80]. Thus, when the load of a specific model service increases, we are able to find spare GPUs from other models to scale up the serving capability of this model.

Autoscaling an instance requires two basic steps: (1) initialize a proper execution context, e.g., create CUDA contexts (control plane) and (2) load the model parameters to the GPUs' memory (data plane). We focus on (2) because (1) can be minimized with recent advances in GPU startup methods like checkpoint and restore [40, 79] and our Rust/C++-based serving platform (see also §6.3). For (2), the state-of-the-art system—ServerlessLLM [29] optimizes the data plane with SSD-optimized parameter loading. Unfortunately, it does not account for the scaling speed required by models. Our measurements in the next section show that SSD-based scaling significantly lags behind applications' requirements.

3 Characterizing Scaling Requirements and Compute Network between Instances

Model autoscaling requires fast data plane. If the data plane speed is not fast enough, during burst period, the requests still violate the SLO due to increased queueing time. Specifically, SLO defines the tolerable end-to-end latency measured from the time a request is sent to the system to the time the inference response is returned. Thus, the latency includes the queueing delays waiting for the scaled instance to be ready for inference.

To characterize how different scaling speeds affect SLO attainments, we implemented a simulator on DistServe [81] that provisions models to all GPUs and applies manual delays based on the simulated speed for modeling different scaling speeds. We set TTFT and TBT SLO based on inference speed of different models following prior works [81]. Specifically, we use 450 ms and 150 ms for Llama3-8B model, and 1250 ms and 200 ms for Qwen2.5-72B model with tensor parallelism degree of 4. §6 describes the detailed evaluation setup.

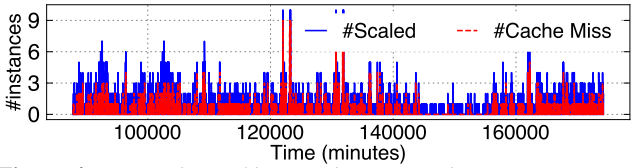


Figure 4: *An analysis of host cache misses when running ServerlessLLM [29] on BurstGPT [71].*

Figure 3 (a)–(d) shows the results: We can see that for a 72 B model, maintaining SLO violations below 60 % requires a minimum per-instance scaling speed of 220 Gbps per-GPU[3], which is only achievable when the model parameters are loaded from the host memory. The scaling time requirement correlates directly with inference time—our evaluated workload (BurstGPT [71]) has an average TTFT of 771 ms (with queueing time). Thus, to achieve 1250 ms SLO for all requests, the scale time must be below 500 ms, so a 576 Gbps per-GPU network speed is required (measured by dividing the parameter size by the scale time). This far exceeds what vendor-provided per-GPU SSDs bandwidth can deliver (2–10 Gbps per-GPU [35, 9, 27], detailed in §A.2).

Loading model parameters from host memory is not effective due to misses. While caching the parameters on the host CPU memory can meet the scaling speed requirement for some setups (e.g., 8 B, 24 B) with the fast host-GPU interconnects (256 Gbps PCIe 4.0), cache misses are common in real-world traces, because the scarce host memory cannot support the caching all models deployed on the MaaS. Figure 4 presents the number of instances scaled and cache misses encountered in the BurstGPT workload using ServerlessLLM [29]. Following its setup, we set a 5-minute keep-alive interval for caching models at the host. The miss rates range from 20–46%, depending on the time, which aligns with the numbers reported in ServerlessLLM's paper (25–60%). Interestingly, many misses occur when scaling multiple instances, because involving more hosts increases the

[2] It also stops a serving instance to scale down. Since scaling down is simpler, we omit its details for brevity.

[3] 72 B model uses four GPUs per-instance.

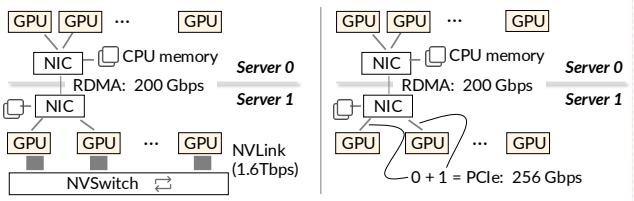


Figure 5: *A illustration of networking in* MAAS *with NVLink (a) and without it. Note that the 256 Gbps PCIe is shared between two GPUs attached to it [48].*

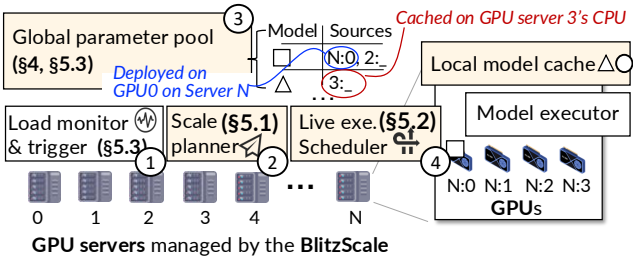


Figure 6: *System architecture of* BLITZSCALE. *New modules introduced by* BLITZSCALE *are marked with* □.

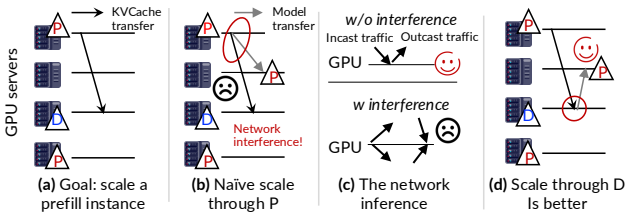


Figure 7: *(a) An illustration of scaling a prefill instance for LLM PD disaggregated serving. (b) Naively scaling from a prefill instance imposes network interference. (c) Interference can be avoided by leveraging the bi-directional feature of modern DCN networking. (d) An improved scale plan with the bi-directional in mind.*

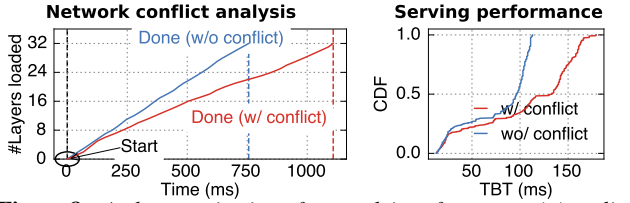


Figure 8: *A characterization of network interference on (a) scaling speed and (b) serving performance.*

probability of scaling a model on a host without the cached parameters. Therefore, we still need accelerating scaling speed when the model parameters are not cached at the host.

Opportunity: fast and underutilized compute network. First, compute networks between GPUs (and CPUs) have comparable or even faster speeds than host-to-GPU link. As shown in Figure 5, the inter-GPU network (RDMA) operates at 200 Gbps, which is close to the host-to-GPU PCIe speed (256 Gbps). With NVLink, the speed is much faster. More importantly, these networks are underutilized during serving. Figure 3 (e) and (f) measure the peak network usage of DistServe [81], a PD disaggregated serving system that heavily utilizes the network due to KVCache transfers. To measure peak usage, we provisioned all the GPUs for serving, and evaluate a workload with the maximal request rate that our clusters can serve. Even under peak load, more than 40% of the network capacity is free, opening up the opportunity to use the compute network for the scaling data plane.

4 System Overview of BLITZSCALE

BLITZSCALE scales models through the compute network to accelerate scaling even under cache misses on the host. We achieve this by first managing model parameters—scattered across GPUs behind serving instances (for deployed models) and CPUs (cached at the host)—through a global parameter manager. The manager maintains a mapping between models and their sources. With the manager, we can quickly read parameters from these sources with the fast RDMA or NVLink. Besides, we also offload computation from overloaded instances to instances with partially loaded parameters to achieve live scaling.

System architecture and workflow. Figure 6 shows our system architecture. Like prior work [56, 29, 62], we have a load monitor (①) tracking the serving load for each model service, and deciding whether to scale and how many new instances are required (§5.3). On each machine, we further adopt off-the-shelf GPU kernels FlashInfer [1] to query the model efficiently. The key differences are twofold. First, our scale planner (②) will derive a scaling plan that guides how to load parameters onto the scaled instances (§5.1) with compute network efficiently. The planner consults the global parameter manager (③) to identify the sources of model parameters. In our example, the new instance can load the parameters of model □ from the GPU0 on host N $(N, 0)$, or from host 2's CPU memory $(2, _)$. Second, during scaling, our live execution (exe.) scheduler (④) will redirect requests between instances to fully utilize the scaled instances even before the parameters are fully loaded (§5.2).

Challenges and approaches. Despite leveraging fast networking, making autoscaling fast and live needs to address the following challenges.

C#1. Online interference-free scale plan generation. Generating the scale plan is similar to generating a *multicast* plan [18, 22, 32, 16, 17], i.e., how to quickly distribute data (parameters) from some sources to targets. There are two additional requirements for autoscaling. First, the plan must be generated online on dynamically changing sources and targets, but optimal plan generation is NP-hard [18] on a heterogeneous network. Second, the plan needs to eliminate interference between loading and the serving workload. Figure 7 shows an example when a model is served via PD disaggregation. In PD disaggregation serving workload KVCache is migrated from prefill to decode instances (a), and this migration overhead can be hidden [55]. However, sup-

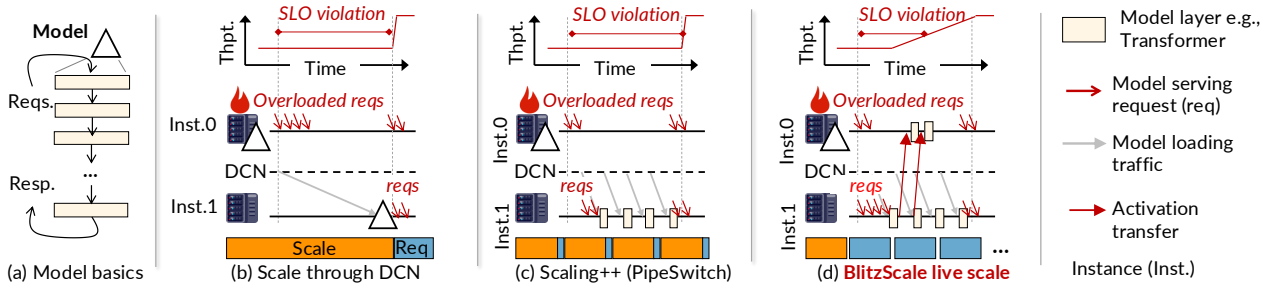


Figure 9: *(a) An overview of how model executes requests. (b) An illustration of naive model scale through DCN. (c) An illustration of an optimized scaling method with overlapped execution [15], but it still cannot be live. (d) How* BLITZSCALE *scales models lively.*

pose we want to scale a prefill instance: if we naively select a prefill instance as the source (b), the scaling will compete the network bandwidth with the serving workload, leading to 1.5 × longer scale time as well as 50% tail TBT increase due to the amplified KVCache migration overhead (Figure 8 (b)).

To this end, we design a serving-guided greedy plan generation method based on three observations (§5.1). First, the network heterogeneity mainly comes from NVLink, whose speed is extremely fast, i.e., it can broadcast a Llama3 8B to 8 GPUs within 120 ms. Thus, we can abstract instances linked with NVLink as a logical instance group to eliminate NVLink from the network topology. Second, loading parameters from the network is bandwidth-intensive, so we can greedily construct serial forwarding chains [66] for multicast, which is optimal in the common case. Finally, the network (RDMA) between GPU servers is bi-directional [72, 52], meaning that the network flows of incast and outcast don't interfere (c). Thus, we can leverage this feature to avoid interference by removing flows in the same direction on the same network link during plan generation. For example, we can load the parameters from the decode instance to the prefill instance (see Figure 7 (d)).

C#2. Realizing live autoscale. Live autoscaling—allowing the scaled instance to increase system throughput before all parameters are fully loaded—is necessary because SLO violations can still happen (Figure 3 (c)) even with fast networking. It is challenging to achieve this in existing systems. For example, PipeSwitch [15] and DeepPlan [42] leverage the layer-by-layer execution nature of models to perform inference: As shown in Figure 9 (c), once the first layer is loaded on inst.1, they redirect the overloaded requests to it for execution. Meanwhile, inst.1 will load subsequent layers concurrently. However, such overlapping is not live because, until all the layers are loaded, inst.1 still cannot finish requests to increase the system throughput.

To this end, we propose a novel cooperative execution scheme for live autoscaling. The key observation is that though the scaled instance alone cannot finish the requests until all layers are loaded, it can alleviate the load of the overloaded instance with its loaded layers, thus improving the serving throughput. Figure 9 (d) illustrates this. When inst.0 becomes overloaded and inst.1 is under scaling, after inst.1 has loaded the first layer, we redirect all requests from inst.0 to inst.1 for execution. Once inst.1 completes the first layer's execution, it forwards the activation back to inst.0 to process the remaining layers, and the system throughput increases with reduced queued latencies, as queued requests are processed faster. To see why the throughput increases, consider serving a 7-layer model. inst.0 alone will have a throughput of 1/7. With our live scaling, after loading one layer on inst.1, inst.0 only needs to execute 6 layers, so its throughput increases to 1/6. The throughput continues to improve as more layers are loaded, reaching the peak (doubled) after half of the layers have been loaded—half of the scaling time. §5.2 describes our ZigZag scheduling for coordinating overloaded and new instances during live autoscaling to achieve optimal performance for live autoscaling.

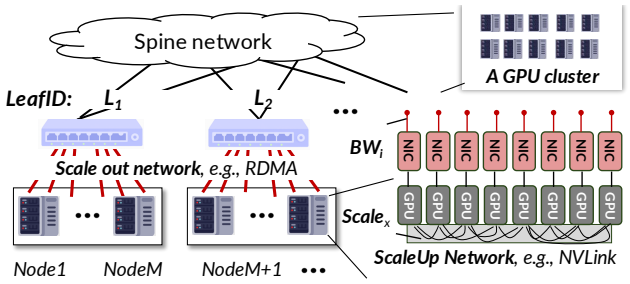


Figure 10: *An illustration of how* BLITZSCALE *models the network.*

5 Detailed Design and Implementation

5.1 Online network-based scale plan generation

When the planner is notified to scale the parameters onto n new GPUs, it will get s sources from the parameter pool, find t spare GPUs as target and generate a plan on how to send parameters from s sources to a subset of n GPUs in t targets. There are three metrics to minimize for the generation: (1) the scale time, (2) the plan generation time, and (3) the interference with serving workloads.

Modeling the network between GPUs. Effectively generating a plan requires a model of the network between sources and targets, which is non-trivial due to the complexity of the network topology in serving clusters. Our model assumes a

Input: D_{src}, source GPUs; D_{tgt} target GPUs; n: number of GPUs to scale;
BW_i: return the scale out network bandwidth of GPU i;
L_i: return the leaf switch ID of GPU i;
$Scale_i$: the scaleup domain ID of GPU i.
Output: **Plan**, a graph indicating how the parameters are loaded.

```
1  D_src = D_src.prune().group_by(L_i).sorted_by(sum([BW_i])).flatten()
2  D_tgt = D_tgt.group_by(Scale_i).sorted_by(sum([BW_i]))
                ▸ sort D_tgt according to the order of Leaf ID in D_src
          .groupby(L_i).sortby(D_src.index(L_i).min()).flatten()
3  m = 0; Plan = ∅
4  While not D_tgt.empty() and m < n:
   |  ▸ G_tgt: targets group connected by the scale up network
   |    L_tgt: the leaf switch of these targets.
5  |  G_tgt, L_tgt = D_tgt.pop_front()
6  |  If D_src.filter(= L_tgt).sum([BW_i]) ≥ G_tgt.sum([BW_i])
   |  |  ▸ sources within the leaf have sufficient bandwidth for loading
7  |  |  D_tmp = D_src.truncate(= L_tgt); D_src ++ D_tmp
8  |  G_src = D_src.truncate(sum([BW_i]) ≥ G_tgt.sum([BW_i]))
9  |  Plan = Plan ++ (G_src, G_tgt); m = m + |G_tgt|
10 |  D_src = G_tgt ++ D_src
11 return Plan
```

Notation:

- **++** ▸ join 2 data collections, e.g., [x, y] ++ [z, w] = [x, y, z, w]
- **[Func]$_i$** ▸ apply **Func** to all elements of a data collection, e.g., G.sum([BW$_i$]) is the summation of bandwidth of all elements in G
- **Iter. index(value)** ▸ locations where a unique value first occurs in **Iter**
- **Iter.truncate(predicate)** ▸ pop front until the first remaining element satisfies predicate
- **Iter.truncate(flatMap, predicate)** ▸ pop front and apply flatMap onto the popped elements until the result satisfies predicate

Figure 11: *The pseudocode of the plan generation algorithm.*

simplified scale-up and scale-out network hybrid networking, widely adopted for GPU clusters [25, 70, 50]. Figure 10 (a) illustrates the modelling.

First, we model the GPUs connected via fast scale-up networking like NVLink as groups of GPUs. Such GPUs have ultra-high interconnect bandwidth (1,600-3,600 Gbps) so scaling within a group has negligible overhead. On the other hand, GPUs connected via slower scale-out networking like RDMA are more difficult to model due to a hierarchical structure. To this end, we adopted a simple leaf-to-spine model that covers widely deployed topologies including Clos and Rail-optimized [70] with different subscription ratios: each GPU (i) has a BW_i bandwidth connected to a leaf switch (LeafID), where GPUs within the same leaf switch have a full-mesh connection, i.e., the bandwidth between GPUs (i) and (j) is $\min(\mathrm{BW}_i, \mathrm{BW}_j)$ with full bandwidth. Second, leaf switches are connected to spine switches, with inter-leaf bandwidth equal to or smaller than the intra-leaf bandwidth. For simplicity, we don't model the spine network and rely on upper-tier protocols like Virtual Link Trunking (VLT [30]) and Equal-Cost Multi-Path (ECMP [37]). to support efficient inter-leaf networking.

Multicast-chain-based greedily plan generation. To quickly generate the plan online, we use a three-step greedy

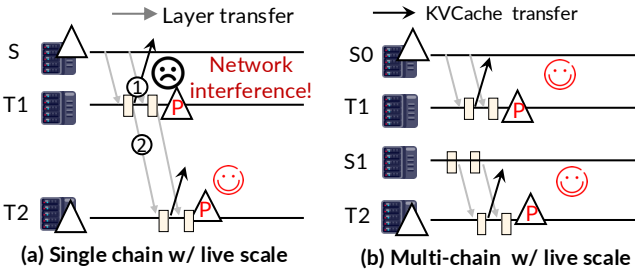


Figure 12: *An illustration of why multiple chains are better especially under live scaling.*

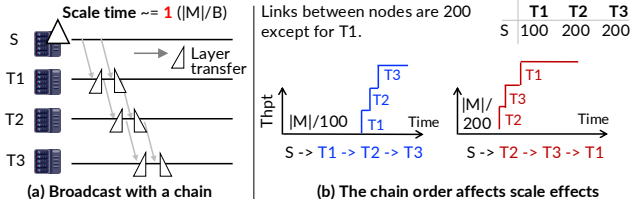


Figure 13: *An illustration of (a) why chain is friendly to broadcast with huge bandwidth requirement and (b) why we chooses a specific chain order. $|M|$ is the model size and B is the slowest network bandwidth between nodes in a chain.*

algorithm as shown in Algorithm 11. First, we prune the sources to avoid any interference with serving workloads (Line 1). Second, we group targets connected with scale-up networking like NVLink as a group (Line 2) such that we can leverage the NVLink broadcast to efficiently realize parallel sharded parameter transfer, see Figure 14 and described below. Finally, we form multiple serial broadcast chains (Line 3–10) to generate the plan.

Specifically, a serial broadcast chain is formed by a set of source and target nodes, i.e., $S \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$. Note that a node in a chain may have multiple GPUs. Such a chain has a nice property that it is optimal in bandwidth-intensive transfer like model scaling, because the overall transmission time is irrespective of the instances scaled with such a chain. As shown in Figure 13 (a), when T1 receives the first layer, T1 immediately forwards it to T2. Meanwhile, S will continue to send the second layer to T1, so the time of sending the first and second layer is overlapped.

While a serial chain is sufficient for efficient parameter broadcasting for nodes that are connected with the same bandwidth links, multiple chains are necessary in a leaf-spine network where inter-leaf bandwidth may vary and in our live scale setup. This is because (1) multi-chain avoids relatively slow inter-leaf communications if each leaf switch has sources and targets (Line 6–7), and (2) it enables more interference-free live scaling especially in PD disaggregation setting. Figure 12 illustrates the latter: suppose we want to scale up two prefill instances in a live manner, the KVCache will be transferred to the decode instances once prefill is done. With a single chain, only $T2$ can join the live scale without network interference, because at $T1$, the KVCache transfer (①) interferes with the parameter forward traffic (②). With

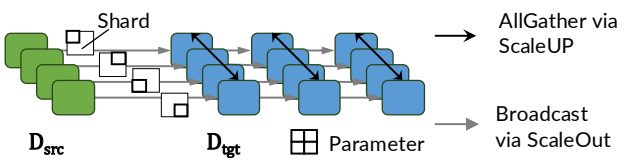


Figure 14: *An illustration shareded parameter transfer with scale up network.*

two chains backed by two parameter sources (b), both $T1$ and $T2$ can live scale without interference.

Note that the order of nodes in the chain is important: we chose a decreasing order with respect to the aggregated link speed between nodes (Line 2, 5). This is because sending to nodes with higher bandwidth achieves a faster increase in the serving throughput. As shown in Figure 13 (b): suppose the source (S) sends parameters to $T2$ twice as fast as $T1$. A chain order of $S \rightarrow T2 \rightarrow T1$ is better than $S \rightarrow T1 \rightarrow T2$ because the downtime of $T2$ is only half. Note that the source can be a group of GPUs because GPUs typically have dedicated network cards in our setup.

Optimization: parallel sharded parameter transfer from multiple sources. For a broadcast chain where the source and target contain GPUs with duplicated parameters, we further leverage the scale-up network to parallelize a transfer link. Figure 14 shows a concrete example. Suppose the source and target nodes have four GPUs each. For such a transfer, each source GPU only needs to forward 1/4 of the sharded parameters to each target GPU, where the target GPUs can use NVLink-based AllGather to get the full parameters. This reduces the scaling time to 1/4 as the NVLink AllGather time is negligible.

5.2 Efficient live autoscaling with ZigZag scheduling

Selecting instances for live scaling. After getting the chains from Algorithm 11, we select instances to participate in live autoscaling based on the following criteria: (1) the necessity of live autoscaling, i.e., when a stop-the-world scaling will cause SLO violation and (2) the presence of overloaded instance that can cooperate. Both are readily available: (1) we can profile the relationship between load speed and SLO violation in Figure 3 for the judgement and (2) autoscaling is typically triggered when the system is overloaded. Thus, for each overloaded instance, we will identify an instance in the chain that satisfies (1), typically the tail instances in the chains as it has the slowest link.

Live autoscale protocol with paired instances. Suppose we have selected a new instance (inst.1) to offload computations from an overloaded instance (inst.0). To begin live autoscaling, we use a three-step transition protocol: (1) Once inst.1 starts loading parameters, we redirect all queued and new requests from inst.0 to it for execution. The redirection time is negligible because the request payloads are much smaller than the model. (2) After the first layer is loaded on inst.1,

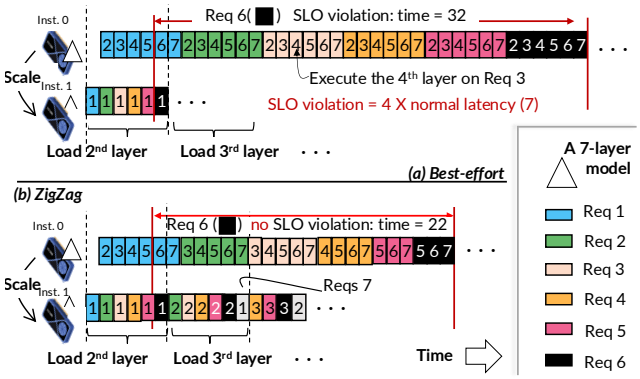


Figure 15: *An illustration of the necessity of ZigZag scheduling. Note that the execution starts when the first layer has been loaded to instance 1 (inst.1). Our example assumes the time of loading a layer can perform 6-layer computations.*

it begins executing the first layer of all requests. Note that during the loading of the first layer, inst.0 remains active by processing its pending requests. Finally, when the model has completed loading on inst.1 (3), requests will be re-distributed evenly between both instances.

The scheduling problem. A key issue to address in the above (3) is how to best utilize inst.1 to maximize the goodput during live autoscaling. Specifically, we should decide a pipeline configuration for each request batch, i.e., how many layers to execute on inst.1 and inst.0, respectively. One naive policy is best-effort: for each batch, we execute as many layers as possible on inst.1 (not exceeding half) and execute the rest on inst.0. While it adapts configurations with model loading, we found it is suboptimal because inst.1 has limited serving capacity during initial loading, so most requests are still queued at inst.0, causing SLO violations. Figure 15 (a) shows a concrete example of a 7-layer model executed with the best-effort scheduling. The load time of one layer can do 6-layer computations, a common setup (e.g., Llama2-7B model with a moderate batch size of 2000 prefill tokens under 200 Gbps RDMA network). Thus, before the second layer has been loaded to inst.1, the current request batches (req 1–6) can only use a $(1, 6)$ pipeline configuration. However, request 6 will suffer from SLO violation due to waiting for requests 1–5 to be completed on inst.0, as their execution time has reduced a little due to the imbalanced loads.

The ZigZag scheduling. To address this issue, our observation is that by delaying request scheduling on inst.0, inst.1 will have more layers coming, opening opportunities to balance the workload between instances. Figure 15 (b) shows this: After requests 2–5 have been executed on inst.1, we delay their execution on inst.0 and wait for the second layer to come. This allows us to adopt a more aggressive pipeline configuration $(2, 5)$ for them. Note that the delay won't waste GPU because we can schedule pending requests (e.g., 6). After the second layer has been loaded, inst.1 can come back (thus, in a ZigZag way) to execute the second layer of re-

quests 2–5. Thus, the second layer execution of requests 3–6 is overlapped with the execution of layers 3–6 for request 2. Thanks to this overlap, the overall inference time of request 7 decreases from 32 to 22, now within the SLO.

The above ZigZag scheduling can be formulated as follows. Assuming a first-come-first-serve (FCFS) scheduling policy, the de facto for serving [44, 75, 77]. For ease of presentation, we first assume non-LLM and then extend to LLM in §5.4. The scheduling has two parts:

(1) Pipeline configuration. Given N request batches with equal execution time, we first determine the pipeline configuration $(\mathrm{T}_i, \mathrm{S}_i)$ for them, where T_i and S_i are the number of layers to be executed on the target and source GPU for request i, respectively. The goal is to minimize the average latency, which can be formulated with the following Integer Linear Program (ILP):

$$\text{Latency}_{\text{avg}} = (\sum_{req=1}^{N}\sum_{i=1}^{req}\mathrm{S}_i)/N$$

To see why such a formula holds, consider the example in Figure 15 (b). Each request's latency is the time the source instance finishes its part of the computation, which includes its own execution time and the sum of its previous requests' time (queueing time). We only need to consider previous requests because they are executed in a FIFO order. In our example, request 3's latency is 17 (12 for requests 1 and 2's execution and 5 for its own). For non-LLM, the execution time of each layer is the same if the batch size is the same. Note that we omit the activation transfer latency since it is negligible.

The problem has the following constraints:

$$\begin{aligned}
\min\quad & \text{Latency}_{\text{avg}} \\
\text{s.t.}\quad & \mathrm{S}_i + \mathrm{T}_i = L, \quad \forall i \quad \text{Pipeline limit } \textbf{(C1)}, \\
& \sum_{j=1}^{i}\mathrm{T}_j \le \sum_{j=1}^{i-1}\mathrm{S}_j, \forall i > 1 \quad \text{Pipeline dependency } \textbf{(C2)}, \\
& \text{Time}_l * \mathrm{T}_i \le \sum_{j=1}^{i-1}\mathrm{T}_j + (N-i+1)\times(\mathrm{T}_i - 1), \\
& \forall i > 1 \quad \text{Load limit } \textbf{(C3)}
\end{aligned} \tag{1}$$

C1 ensures that the pipeline should be fully executed. **C2** states pipeline dependency: when the source instance executes request i's S_i layers, the target instance must finish the execution of T_i. The start execution time of request i on the source is $\sum_j^{i-1}\mathrm{S}_j$. The finish time of i on the target instance is $\sum_j^{i-1}\mathrm{T}_j + \mathrm{T}_i$, which simplifies to $\sum_j^{i}\mathrm{T}_j$. Finally, **C3** ensures that once the target instance request's i's T_i layers, all these layers must be loaded, where Time_l is the time to load one layer normalized to the execution time of one layer in pipeline. The term $(N-i+1)\times(\mathrm{T}_i-1)$ indicates that the load time can be overlapped with executing of the succeeding requests of i.

Input: **Q**: an atomic distributed priority queue that stores the requests to be scheduled. The priority is defined as follows: for **req i** and **req j**, the **P(i) > P(j)** if and only if **i < j** and **i** have loaded layers unexecuted. The subscription (e.g., **i**) indicates the request's arrival time.

At New Instance (target instance)

```
0  spawn(update_Q_when_layers_come) ▸ Run in the background
1  while execute under live:
2  |   q = Q.get_front()
3  |   forward_one_layer(q)
```

At Old Instance (source instance)

```
4  while execute under live:
   |   ▸ Pull the pending requests with
5  |   If q, activation = pull_the_earlist_request(Q):
6  |   |   forward_all(q, activation)
```

Figure 16: *The pseudocode of the ILP-free ZigZag scheduling.*

While solving this ILP is NP-hard, it remains manageable (less than 40 ms to solve for Llama3-8B) because models typically have only a few dozen layers. Additionally, we only need to configure the pipeline for the batches of requests executed during parameter loading, which is a dozen of so in practice. Nevertheless, to further eliminate the solving time for models with more layers (e.g., 80 layers for Qwen-72B), we also derived an ILP-free method that we described below.

(2) Scheduling requests in an ILP-free ZigZag way. Specifically, we found that by delaying sending the requests on the source instance and letting the target instance execute the requests once it is free, we can achieve ZigZag scheduling without solving the ILP. Figure 16 shows the pseudocode of how we schedule the requests on both instances. The new instance maintains a priority queue (that can be pulled by the source instance via RPC) for all requests, where the priority is defined by (1) the FCFS order and (2) requests with next-to-execute layer loaded coming first. Once the new instance has executed one layer (Line 3), we keep the executed request in the queue so it can be scheduled back once more layers are loaded. The requests are scheduled on the source instance only if it is not overloaded, i.e., has no pending requests (Line 5). Thus, if the source instance is busy, the request will still be executed on the target instance.

5.3 Global parameter pool and scaling policy

Global parameter pool and local memory cache. Our global parameter pool tracks the locations of the model parameters across deployed GPUs and host CPUs with local memory cache. To ensure at least one copy of the model parameters is available in the memory of GPU or host at the cluster scale, during system initialization, we distribute one copy of the model's parameters evenly to the CPU hosts and track their locations at a centralized manager. When a model is deployed to or reclaimed from a GPU, we further update the locations in the manager, and reclaim/reload cached copies on the host cache.

Scaling policy. Our paper focuses on the autoscaling mechanism, which is orthogonal to the autoscaling policies, includ-

ing collecting workload metrics with workload monitoring and determining how many new instances to scale based on these metrics. Our current implementation follows prior works [62, 2] that first records the serving loads with tokens per second and KVCache usage globally.

For scaling up, when the average monitored load surpasses a pre-defined upper bound, we allocate sufficient instances to meet that demand. The upper bound can be derived by profiling the average serving load per-instance offline. We leave a more detailed explanation in another paper. For scaling down, we follow previous works'[56, 29] timeout-based policy: when the average monitored load falls below a lower bound in a time window, we shut down some instances and revoke all GPUs assigned to them. Given BLITZSCALE's rapid autoscaling capabilities, we adopt an extremely short sub-second level timeout.

5.4 Specializations and optimizations for LLM

While most techniques described above work for all models following a layer-by-layer architecture, the unique characteristics of LLMs especially for LLMs served with PD disaggregation require several specializations and optimizations.

Retrofitted live pipeline scheduling formulation. Our formulas and constraints described in §5.2 cannot be directly applied to LLMs because the prefill and decode time of a layer is approximately linear to the total batched token size [55, 75]. For prefill-only live scheduling, e.g., autoscaling a prefill instance in PD disaggregation, we fix the formulation by adding a regulation parameter for each request batch by profiling its execution time with the counts, similar to a priori work [75]. A more tricky case involves handling decoding, e.g., when scaling instances that combine prefill and decode, or scaling a decode instance in PD disaggregation. The complexity arises because decode batch size changes dynamically due to its auto-regressive nature. Fortunately, our ILP-free scheduling method can also work for decoding.

Supporting PD colocation. We seamlessly support PD colocation since a PD-colocated instance is a normal model instance. Meanwhile, our ILP-free ZigZag scheduling also applies to pipelined execution under PD colocation [5].

Live scaling decode instances in PD disaggregation. Live scaling a decode instance in PD disaggregation without interference is impossible due to the incast bandwidth contention of both parameter loading and KVCache transfer. Thus, we leverage the fact that the prefill and decode instances share the same model parameters, so we can live scale a decode instance by first mutating some prefill instances to decode instances, while concurrently live scaling the prefill instances to compensate for the prefill throughput.

Optimized scaling policy for PD disaggregation. Prescaling instances can hide the cost of scaling, but a too-early scaling wastes GPU resources. In PD disaggregation, we found we can pre-scale decode instance at zero cost, because the need for scaling decode instances can be evidenced by the requirement for scaling prefill instances. Specifically, once we found a significant requirement for scaling prefill instances, we will simultaneously scale decode instances. This effectively hides the scaling cost of decode instances, and is even effective for other systems like ServerlessLLM [29], see §6.1.

	Cluster A (m x g)	**Cluster B** (m x g)
GPU	A800 80 GB (4x8)	A100 80 GB (2x8)
GPU-GPU (intra)	1.6 Tbps NVLink	256 Gbps PCIe
GPU-GPU (inter)	100 Gbps RDMA	100 Gbps RDMA
Host-GPU	128 Gbps PCIe	128 Gbps PCIe
SSD-GPU	10 Gbps	10 Gbps

Table 1: *Evaluation clusters. m is the number of hosts and g is the number of GPUs per host.*

6 Evaluation

System implementation. BLITZSCALE is a MAAS system capable of serving both traditional models and LLMs with 24,000 lines of Rust and C++ code. It builds upon widely applied LLM optimizations like PD disaggregation and continuous batching. We leverage existing highly-optimized serving system components (with no autoscaling support) wherever possible. For instance, all our GPU kernels for LLM come from FlashInfer [1]. We choose a native-language-based framework implementations because we found it is challenging to implement fine-grained scheduling in Python. §A provides more implementation details.

Testbed. Our evaluations are conducted on two testbeds listed in Table 1. Cluster A can serve larger models (e.g., 72 B) with tensor parallelism [44] thanks to the NVLink while cluster B is more suitable for serving single-GPU models.

Evaluated traces and models. Because the scaling requirements are closely related to the incoming request rates, we chose three typical real-world traces: BurstGPT [71], AzureCode and AzureConv, both from Azure [55]. The detailed trace shapes are shown in the first column in Figure 17. Since the traces are collected from clusters with different serving capabilities, we follow the standard approach [46, 8, 36] to scale the traces to fit our clusters. Specifically, we scale the trace with temporal pattern preserved using TraceUpscaler [57], and the scaled average request rate is half of the maximum serving capacity of our cluster.

For models, we focus on evaluating LLMs because other non-LLMs are much smaller and trivially scale efficiently with BLITZSCALE. Specifically, we choose Llama3-8B, Mistral-24B and Qwen2.5-72B, all are popular LLM models with high accuracy. Since BLITZSCALE is only sensitive to the model size, we may omit the detailed model family name and only uses their sizes in the following description for simplicity. For small model (8B), it only needs one GPU

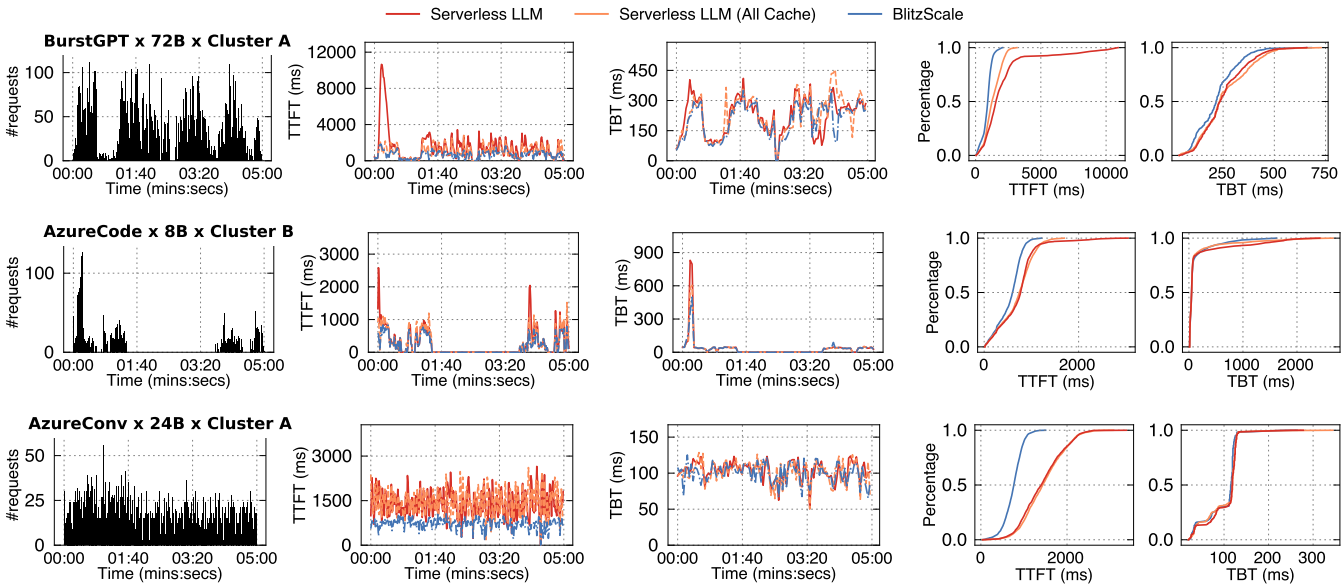


Figure 17: *End-to-end performance comparison between* BLITZSCALE *and ServerlessLLM on various workloads, models and clusters.*

per instance while for 72 B models the minimal number of GPUs used by one instance is 4.

Comparing targets. Without explicit mention, we compare BLITZSCALE with the following baselines:

1. ***ServerlessLLM (S-LLM)*** ***[29]*** is the state-of-the-art MAAS with a focus on accelerating autoscaling speed. It utilizes host memory to cache recently loaded models with a time-to-live eviction policy. Under cache misses, it loads parameters from SSD with SSD bandwidth fully utilized.
2. ***ServerlessLLM optimal (AllCache)*** is the autoscaling speed optimal version of ServerlessLLM that always loads the parameters from the host cache.
3. ***DistServe*** [81] is the state-of-the-art LLM serving system without autoscaling support. It leverages PD disaggregation. We chose it because autoscaling is more challenging in PD disaggregation due to the complexity of multiple instances scaling (prefill and decode) and the need to avoid scaling interference. We compare with other common PD colocation systems like vLLM [44] in §6.4.

For a fair comparison, we adopted the same scaling policy for both BLITZSCALE and variants of S-LLM.

6.1 Autoscaling performance under real-world traces

Due to space limitations, for each model, we choose one trace on one of the clusters to evaluate the performance. Figure 17 presents the end-to-end performance when serving with a prefill and decode disaggregation setup where the instances for different phases are scaled independently. The first column shows the request rate of the trace, the second and third columns show the mean TTFT and TBT, respectively, where each point is the average latency measured during a small time window (1s), and the final two columns present the cumulative distribution function (CDF) of the TTFT and TBT during the evaluation period, respectively. We focus on comparing with S-LLM and AllCache in this section and leave the comparison with DistServe in the next section, as it does not support autoscaling.

Overall performance. First, we can see that BLITZSCALE has the lowest TTFT and TBT in all workloads thanks to the fast autoscaling speed. Specifically, on BurstGPT, the TTFT is 75.5 % and 21.1 % shorter than S-LLM and AllCache, respectively, and the TBT is 7.4 % and 5.1 % shorter, respectively. Nevertheless, the degrees of improvement are different across metrics due to the unique characteristics of the prefill and decode phases. Meanwhile, the behaviors of systems, especially S-LLM, are different across workloads due to the different request arrival patterns (see the first column). We elaborate on the differences in the following.

TTFT vs. TBT. BLITZSCALE is more effective in reducing the TTFT than TBT on all workloads. This is due to two reasons. First, the decode instance can be pre-scaled thanks to our optimized policy (§5.4), which we apply to all baselines. Specifically, when the prefill throughput increases, BLITZSCALE (and its baselines) will simultaneously scale the decode instances, yet no more decode instances are needed at the scale time. Thus, the scaling time is overlapped with the prefill time, which hides some scaling overheads. Second, decode scales less than prefill because as long as there is sufficient memory on the decode instances, all systems can handle decoding with a slightly increased TBT due to no queueing. Since all models adopt modern LLM optimization group query attention [6] with low memory footprint, decoding instances are more sufficient than prefill instances. Nevertheless, BLITZSCALE still achieves a 5.1–7.4 %, 88.3–

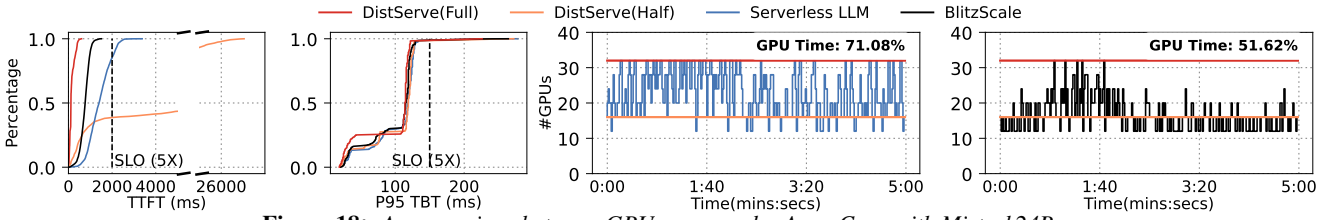


Figure 18: *A comparison between GPU usage under AzureConv with Mistral 24B.*

94.1 % and 0.7–1.8 % shorter TBT than S-LLM and AllCache on three workloads, respectively.

Comparisons between different workloads results. BLITZSCALE always outperforms AllCache thanks to fast network-based autoscaling as well as live scaling, but S-LLM has different behaviors compared to AllCache in these workloads. On BurstGPT, S-LLM first has a sharp TTFT spike at the first burst (time 0:05), while it is close to AllCache in future bursts, because future bursts can benefit from host cache. In comparison, on AzureCode, S-LLM has spikes under both bursts (time 0:05 and time 03:25), because the gap between two bursts makes the host cache evicted due to a time-to-live policy. Finally, on AzureConv, since the bursts continuously arrive, S-LLM always hits the host cache, so the performance—see the CDF graphs—is similar to AllCache.

6.2 Performance and resource usage

Comparison with non-autoscaling systems. We first compare BLITZSCALE with DistServe. Since DistServe does not support autoscaling, its performance is highly dependent on the number of provisioned instances. Therefore, we evaluate two setups: DistServe (full) uses all GPUs in our cluster and represents an optimal performance at the cost of GPU waste. On the other hand, DistServe (half) uses GPUs with the average number of instances required to handle all the workloads within the evaluation period. For simplicity, we only present the results on AzureConv on 24B models, the overall trends are similar. We have carefully calibrated DistServe's performance, such that when autoscaling is disabled in BLITZSCALE, DistServe has the same performance as BLITZSCALE in all setups.

The first two columns of Figure 18 present the latency results. First, it can be observed that DistServe (full) has the best performance, because the GPU is over-provisioned so it doesn't suffer from queueing or scaling overhead. Nevertheless, BLITZSCALE still achieves the same service level objective (SLO) as DistServe (full) while S-LLM incurs 18.7 % SLO violations. We follow the traditional 5 × SLO [81] since all our workloads (chat and code generation) are latency-sensitive. Specifically, if a request's end-to-end (TTFT or TBT) latency is exceeds 5 × the average latency, it violates the SLO. Finally, DistServe (half) has the poorest performance: on average, BLITZSCALE has a 95.8 % and 1 % shorter TTFT and TBT than DistServe (half). BLITZSCALE achieves this by using the same GPU time for serving this model as DistServe

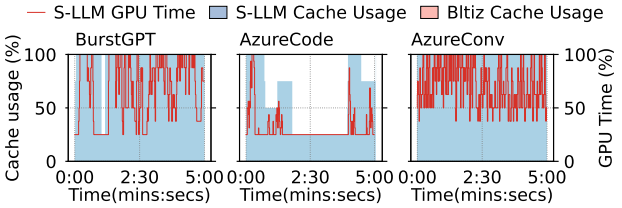


Figure 19: *A comparison of host cache usage on S-LLM and* BLITZSCALE *under the evaluated workloads.*

(half), and this time is 50 % smaller than DistServe (full), which we elaborate next.

GPU time used. The last two columns of Figure 18 show the GPU time used by S-LLM and BLITZSCALE, respectively. For S-LLM and BLITZSCALE, we collected the aggregated GPU usage at each time point for both prefill and decode instances, and the overall time is calculated by integrating the area under the curve. For variants of DistServe, their GPU time is constant across the evaluation period. We can see that BLITZSCALE has 19.46 % lower GPU time than S-LLM thanks to the fast autoscaling capability: with low scaling speed, there would be more queued requests, so the system would trigger more scaling operations that use more GPU time. This is unnecessary with BLITZSCALE. Even with less GPU time used, BLITZSCALE has a 48.1 % and 1.8 % shorter TTFT and TBT than S-LLM, respectively.

Host cache usage. Compared to ServerlessLLM, BLITZSCALE also consumes less host memory for parameter caching. Figure 19 reports the host memory usage for different systems. We omit AllCache and DistServe, as AllCache always fully replicates parameters to all hosts while DistServe does not need caching. We normalize the host cache usage as different workloads use different clusters. The results deliver two messages. First, BLITZSCALE only needs minimal host caching (less than one) to achieve fast autoscaling: this is as expected by our design because we prefer to load parameters from GPUs of instances that serve the model, and even when no serving instance is available, we only need one host copy due to the network-based multicast. Second, the memory usage of ServerlessLLM is proportional to the number of hosts involved in the serving, so a model can quickly "pollute" the host cache. This is non-optimal for an MAAS system because it can simultaneously serve many models while the host cache is limited.

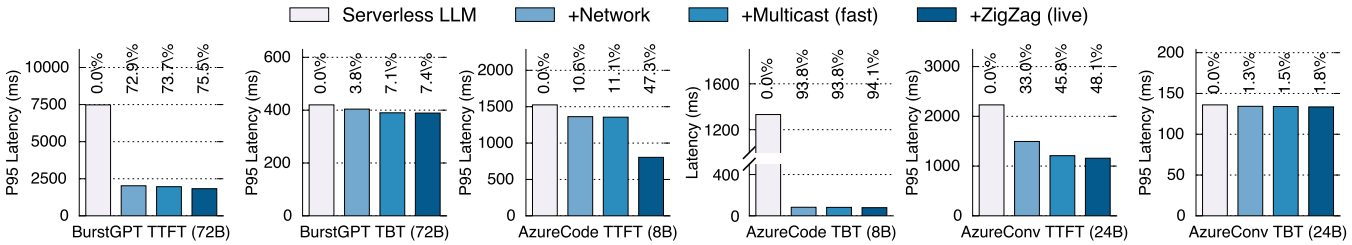


Figure 20: *An ablation study on the effectiveness of our proposed techniques.*

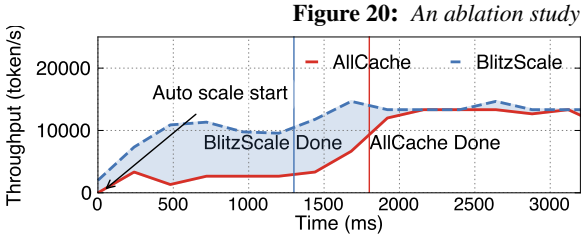


Figure 21: *A detailed look at how* BLITZSCALE *and AllCache scale a 24 B model to 6 prefill instances on cluster A.*

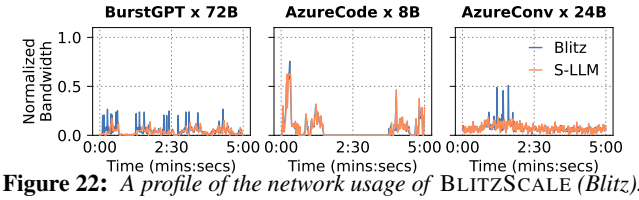


Figure 22: *A profile of the network usage of* BLITZSCALE *(Blitz).*

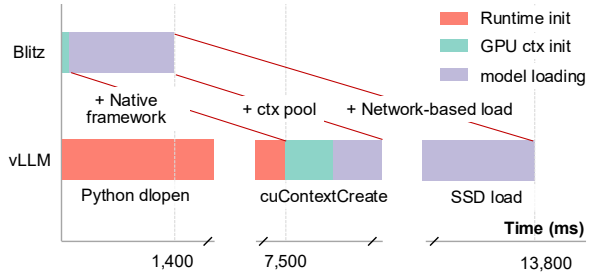


Figure 23: *A comparison of init time of* BLITZSCALE *and vLLM*

6.3 Detailed performance analysis

A detailed look at the live scale. Figure 21 shows a throughput timeline when using BLITZSCALE and AllCache to scale six 24 B prefill instances. BLITZSCALE utilizes two broadcast chains (each involving 3 instances), while the end instances involve a live autoscale. The start instances are the decode instances. For AllCache, it directly loads the parameters from the host memory of the scaled instances. We can see that first, even with a few loaded layers (e.g., at time 500 ms), BLITZSCALE can gradually emit tokens as a result of live execution. Second, BLITZSCALE can scale faster even compared with AllCache, thanks to our NVLink-based fused link transmission protocol: it can finish scaling in 1,200 ms while AllCache takes about 2,000 ms.

Ablation study. Figure 20 conducts an ablation study on the effectiveness of our proposed techniques. We measured the effectiveness by incrementally enabling different techniques and reporting the results on the three workloads: "+Network" leverages fast computing network instead of SSD for autoscaling, "+Multicast (fast)" further applies our optimized parameter broadcast protocol described in §5.1, while "+ZigZag (live)" enables live autoscaling of §5.2.

First, we can see that all techniques are effective in improving the end-to-end serving performance, but the degrees differ across workloads. First, "+Network" improves the scaling performance in all workloads thanks to the higher bandwidth for the autoscaling data plane. Second, "+Multicast (fast)" is effective in AzureCode and AzureConv, but it is less effective in BurstGPT due to the limitations of our cluster (up to 8 instances can be scaled on 72 B model), so there are no cases to simultaneously scale multiple instances, which is the targeted case for this technique. Live autoscaling is mostly effective in AzureCode because it is evaluated on a cluster with slow networking (Cluster B). Finally, our techniques are not such effective on decoding because decode instances are sufficient in most cases, which we have discussed in §6.1. One exception is AzureCode: in this workload, the prefill throughput increases slower than others (see the first column of Figure 17), so the decode instances are triggered later. As a result, the slow scale of ServerlessLLM's SSD cannot be hidden, making a faster scaling more beneficial.

Network usage. Figure 22 shows the network usage of BLITZSCALE and S-LLM: we can see that though BLITZSCALE leverages compute network for autoscaling and the scale frequency is high (see the last column of Figure 18), the additional network usage is negligible.

Control plane vs. data plane of model autoscaling. Figure 23 compares the control plane and data plane overhead during model autoscaling with vLLM. We can see that with proper optimizations, the control plane overhead is negligible.

6.4 Performance under LLM PD colocation

Finally, Figure 24 compares the performance of BLITZSCALE and vLLM where the serving is conducted in PD colocation on BurstGPT workloads with Llama2-7B model. The general trend is similar to PD disaggregation: BLITZSCALE has comparable performance with over-provisioned vLLM, while compared with an average provisioning, BLITZSCALE has a 0.24× shorter P99 TTFT. Interestingly, we found BLITZSCALE has even shorter tail TTFT compared with over-provisioned vLLM, because our scheduling framework is optimized for cluster serving.

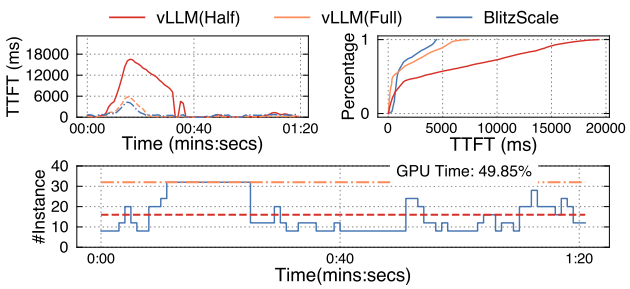


Figure 24: *A comparison of* BLITZSCALE *and vLLM on the BurstGPT workload.*

7 Related work

Optimizing model serving without autoscaling. Serving models at scale is non-trivial. A significant body of research focuses on how to efficiently utilize GPUs to accelerate model serving [81, 75, 44, 45, 77, 55, 31, 68], e.g., Orca [77] proposes iterative-scheduling and selective batching. AlphaServe [45] employs pipeline parallelism to better handle load spikes, but it cannot adjust pipeline instances dynamically. These systems assume running on a fixed pool of GPUs, and we have shown the necessity of dynamically adjusting pool size and how to achieve so efficiently with BLITZSCALE. BLITZSCALE complements these single-instance model serving systems with fast autoscaling mechanisms: we build upon them for fast model serving on a single instance, and additionally provide ultra-fast scaling when the system needs to change the number of serving instances.

Dynamic scaling serving instances. Dynamically scaling serving instances is challenging, mainly because the size of model weights is huge and still increasing, so loading them to the accelerators (data plane) is time-consuming. Some existing works accelerate the loading [15, 41, 46, 62]: For example, both PipeSwitch [15] and DeepPlan [41] leverage the layer-by-layer character of models to overlap the inference and parameter loading to hide the loading cost. They only focus on host-to-device loading and such overlap is not live especially when the models are large. SpotServe [46] and Llumnix [62] realize live migration but migration cannot fully unleash the computing capabilities of both instances. BLITZSCALE provides a new mechanism to scale serving instances lively during parameter loading, resulting in throughput increase even with unfinished loading, which we have shown critical in reducing latencies under bursty workloads.

A concurrent work λScale [78] also focuses on using network to accelerate model autoscaling. The key difference is that λScale scatters the parameters scaling on multiple instances to reduce the time for scaling at the cost of decreased serving throughput, while BLITZSCALE seamlessly scales full parameters on all instances with a similar speed, yet does not sacrifice the throughput thanks to our multicast-chain-based scaling. Moreover, during scaling BLITZSCALE has a gradually increasing throughput thanks to our live scaling while λScale is still a stop-the-world approach.

Accelerating coldstart in serverless computing. Accelerating model scaling builds upon coldstart acceleration in serverless computing [51, 7, 60, 26, 64, 58, 67], which focuses on starting general-purpose computing instances like containers. We built upon these works, e.g., for accelerating container startup time, yet designed efficient network-based live autoscale tailored for model scaling with the domain-specific knowledge of model serving.

8 Conclusion and Future Work

Autoscaling is the key to achieving both high goodput and hardware utilization in model as a service systems, but the performance overhead introduced by current slow and stop-the-world autoscaling significantly limits its effectiveness. In this paper, we first show that the data plane of model autoscaling can be made fast with less than $O(1)$ caching by leveraging network-based model-aware multicast. We next show that the data plane can be made live through model-aware remote execution. Equipped with these two techniques, our system BLITZSCALE has at most 94.1 % better performance and 19.46 % better resource utilization compared to state-of-the-art serving systems with and without autoscaling, respectively. We believe our work demonstrates the potential and practicability of autoscaling-empowered model as a service systems.

While fast and live autoscaling of BLITZSCALE takes a key step toward modern elastic serving systems, several challenges remain. First, during our investigations, we found scaling policies—determining when and how to scale—may also impact system efficiency. The policy depends heavily on workload characteristics, which we leave as future work. Second, BLITZSCALE currently focuses on instance-level scaling, whereas modern models can scale by changing the parallel configuration within an instance, e.g., scaling experts in mixture-of-experts (MoE) models. While BLITZSCALE in principle works for such a setup, we leave the detailed exploration in the future work.

Acknowledgment

We would like to thank OSDI reviewers and our shepherd for their insightful feedback. We sincerely thank Qinwei Yang, Xinhao Luo, and Mingcong Han for giving us feedback on GPU communication library, kernel, driver and runtime. We also thank Xiating Xie, Chenhan Wang and Sundi Guan for refining the presentation of this paper. We thank Alibaba Tongyi Lab for providing the testbed during the early stage of this work. This work was supported in part by the National Key Research & Development Program of China (No. 2022YFB4500700), the Fundamental Research Funds for the Central Universities, the National Natural Science Foundation of China (No. 62202291, 62272291), as well as a research grant from Huawei Cloud.

References

[1] FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer, 2024.

[2] Kubernetes horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, 2024.

[3] Standardized serverless ml inference platform on kubernetes. https://github.com/kserve/kserve, 2024.

[4] Qwen series. https://github.com/QwenLM/Qwen, 2025.

[5] AGRAWAL, A., KEDIA, N., PANWAR, A., MOHAN, J., KWATRA, N., GULAVANI, B., TUMANOV, A., AND RAMJEE, R. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 117–134.

[6] AINSLIE, J., LEE-THORP, J., DE JONG, M., ZEMLYANSKIY, Y., LEBRÓN, F., AND SANGHAI, S. GQA: training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023* (2023), H. Bouamor, J. Pino, and K. Bali, Eds., Association for Computational Linguistics, pp. 4895–4901.

[7] AKKUS, I. E., CHEN, R., RIMAC, I., STEIN, M., SATZKE, K., BECK, A., ADITYA, P., AND HILT, V. SAND: towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 923–935.

[8] ALI, A., PINCIROLI, R., YAN, F., AND SMIRNI, E. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow. 15*, 10 (2022), 2071–2084.

[9] AMAZON. Amazon ec2 accelerated computing instances. https://docs.aws.amazon.com/ec2/latest/instancetypes/ac.html.

[10] AMAZON. Amazon bedrock. https://aws.amazon.com/bedrock/, 2024.

[11] ANYSCALE. Ray serve: Scalable and programmable serving. https://docs.ray.io/en/latest/serve/index.html, 2024.

[12] ARAPAKIS, I., BAI, X., AND CAMBAZOGLU, B. B. Impact of response latency on user behavior in web search. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014* (2014), S. Geva, A. Trotman, P. Bruza, C. L. A. Clarke, and K. Järvelin, Eds., ACM, pp. 103–112.

[13] AWS. Amazon rekognition. https://aws.amazon.com/en/rekognition/, 2024.

[14] AZURE. Azure llm inference traces. https://github.com/Azure/AzurePublicDataset, 2024.

[15] BAI, Z., ZHANG, Z., ZHU, Y., AND JIN, X. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 499–514.

[16] BANIKAZEMI, M., MOORTHY, V., AND PANDA, D. K. Efficient collective communication on heterogeneous networks of workstations. In *1998 International Conference on Parallel Processing (ICPP '98), 10-14 August 1998, Minneapolis, Minnesota, USA, Proceedings* (1998), IEEE Computer Society, pp. 460–467.

[17] BEHRENS, J., JHA, S., BIRMAN, K., AND TREMEL, E. RDMC: A reliable RDMA multicast for large objects. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018* (2018), IEEE Computer Society, pp. 71–82.

[18] BHAT, P. B., RAGHAVENDRA, C. S., AND PRASANNA, V. K. Efficient collective communication in distributed heterogeneous systems. *J. Parallel Distributed Comput. 63*, 3 (2003), 251–263.

[19] CAI, Z., LIU, Z., MALEKI, S., MUSUVATHI, M., MYTKOWICZ, T., NELSON, J., AND SAARIKIVI, O. Synthesizing optimal collective algorithms. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021* (2021), J. Lee and E. Petrank, Eds., ACM, pp. 62–75.

[20] CLOUD, A. Build with generative ai on alibaba cloud. https://www.alibabacloud.com/zh/solutions/generative-ai/build, 2024.

[21] CLOUD, A. The model list. https://help.aliyun.com/zh/model-studio/getting-started/models?spm=a2c4g.11186623.help-menu-2400256.d_0_2.4bb8b0a8ClhJuI&scm=20140722.H_2840914._.OR_help-T_cn%23DAS%23zh-V_1, 2024.

[22] COWAN, M., MALEKI, S., MUSUVATHI, M., SAARIKIVI, O., AND XIONG, Y. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023* (2023), T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., ACM, pp. 502–514.

[23] DEEPINFRA. Custom llms. https://deepinfra.com/docs/advanced/custom_llms, 2024.

[24] DEEPSEEK. Deepep. https://github.com/deepseek-ai/DeepEP, 2025.

[25] DELL. Ai fabrics. https://infohub.delltechnologies.com/en-us/l/dell-technologies-ai-fabrics-overview-1/ai-fabrics-3/3/. 2025.

[26] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 467–481.

[27] ENGINE, V. Volcano engine accelerated computing instances. https://www.volcengine.com/docs/6459/72363.

[28] FACE, H. The ai community building the future. https://huggingface.co, 2024.

[29] FU, Y., XUE, L., HUANG, Y., BRABETE, A., USTIUGOV, D., PATEL, Y., AND MAI, L. Serverlessllm: Locality-enhanced serverless inference for large language models. *CoRR abs/2401.14351* (2024).

[30] G., C. N., ET AL. Virtual link trunking for network configuration and resource optimization. U.S. Patent Application US20160301608A1, oct 2016. https://patents.google.com/patent/US20160301608A1/en.

[31] GAO, B., HE, Z., SHARMA, P., KANG, Q., JEVDJIC, D., DENG, J., YANG, X., YU, Z., AND ZUO, P. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 111–126.

[32] GHAZIMIRSAEED, S. M., ZHOU, Q., RUHELA, A., AND BAYATPOUR, M. A hierarchical and load-aware design for large message neighborhood collectives. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020* (2020), C. Cuicchi, I. Qualters, and W. T. Kramer, Eds., IEEE/ACM, p. 34.

[33] GIGASPACES. Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales, 2024.

[34] GITHUB. Accelerate your development speed with copilot. https://copilot.github.com, 2024.

[35] GOOGLE. Google accelerator-optimized machine family. https://cloud.google.com/compute/docs/accelerator-optimized-machines.

[36] GUJARATI, A., KARIMI, R., ALZAYAT, S., HAO, W., KAUFMANN, A., VIGFUSSON, Y., AND MACE, J. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 443–462.

[37] HOPPS, C. Analysis of an equal-cost multi-path algorithm. Request for Comments 2992, Internet Engineering Task Force (IETF), nov 2000. Accessed: 2025-05-16.

[38] HU, C., HUANG, H., XU, L., CHEN, X., XU, J., CHEN, S., FENG, H., WANG, C., WANG, S., BAO, Y., SUN, N., AND SHAN, Y. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *CoRR abs/2401.11181* (2024).

[39] HUANG, J., ZHANG, M., MA, T., LIU, Z., LIN, S., CHEN, K., JIANG, J., LIAO, X., SHAN, Y., ZHANG, N., LU, M., MA, T., GONG, H., AND WU, Y. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024* (2024), E. Witchel, C. J. Rossbach, A. C. Arpaci-Dusseau, and K. Keeton, Eds., ACM, pp. 421–437.

[40] HUANG, Z., WEI, X., HAO, Y., CHEN, R., HAN, M., GU, J., AND CHEN, H. PARALLELGPUOS: A concurrent os-level GPU checkpoint and restore system using validated speculation. *CoRR abs/2405.12079* (2024).

[41] JEONG, J., BAEK, S., AND AHN, J. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023* (2023), G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, Eds., ACM, pp. 249–265.

[42] JEONG, J., BAEK, S., AND AHN, J. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems* (New York, NY, USA, 2023), EuroSys '23, Association for Computing Machinery, pp. 249–265.

[43] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.

[44] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 611–626.

[45] LI, Z., ZHENG, L., ZHONG, Y., LIU, V., SHENG, Y., JIN, X., HUANG, Y., CHEN, Z., ZHANG, H., GONZALEZ, J. E., AND STOICA, I. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 663–679.

[46] MIAO, X., SHI, C., DUAN, J., XI, X., LIN, D., CUI, B., AND JIA, Z. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024* (2024), R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds., ACM, pp. 1112–1127.

[47] NARAYANAN, D., SHOEYBI, M., CASPER, J., LEGRESLEY, P., PATWARY, M., KORTHIKANTI, V., VAINBRAND, D., KASHINKUNTI, P., BERNAUER, J., CATANZARO, B., ET AL. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), pp. 1–15.

[48] NVIDIA. Doubling all2all Performance with NVIDIA Collective Communication Library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/, 2024.

[49] NVIDIA. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2024.

[50] NVIDIA. Nvidia dgx superpod: Next generation scalable infrastructure for ai leadership. https://docs.nvidia.com/dgx-superpod/reference-architecture/scalable-infrastructure-h200/latest/_downloads/bbd08041e98eb913619944ead1f92373/RA-11336-001-DSPH200-ReferenceArch.pdf#page=8.10, 2024.

[51] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 57–70.

[52] OLUMIDE OLUSANYA AND MUNIRA HUSSAIN. Need for Speed: Comparing FDR and EDR InfiniBand (Part 1). https://dl.dell.com/manuals/all-products/esuprt_software/esuprt_it_ops_datcentr_mgmt/high-computing-solution-resources_white-papers77_en-us.pdf, 2022.

[53] OPENAI. Chatgpt. https://chatgpt.com, 2024.

[54] OPENAI. Creating video from text. https://openai.com/index/sora/, 2024.

[55] PATEL, P., CHOUKSE, E., ZHANG, C., SHAH, A., GOIRI, Í., MALEKI, S., AND BIANCHINI, R. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024* (2024), IEEE, pp. 118–132.

[56] ROMERO, F., LI, Q., YADWADKAR, N. J., AND KOZYRAKIS, C. Infaas: Automated model-less inference serving. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 397–411.

[57] SAJAL, S. M., ZHU, T., URGAONKAR, B., AND SEN, S. Traceupscaler: Upscaling traces to evaluate systems at high load. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024* (2024), ACM, pp. 942–961.

[58] SAXENA, D., JI, T., SINGHVI, A., KHALID, J., AND AKELLA, A. Memory deduplication for serverless computing with medes. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 714–729.

[59] SHAHRAD, M., FONSECA, R., GOIRI, I., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 205–218.

[60] SHILLAKER, S., AND PIETZUCH, P. *FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing*. USENIX Association, USA, 2020.

[61] STABILITY.AI. Activating humanity's potential through generative ai. https://stability.ai, 2024.

[62] SUN, B., HUANG, Z., ZHAO, H., XIAO, W., ZHANG, X., LI, Y., AND LIN, W. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 173–191.

[63] TOGETHER.AI. Inference that's fast, simple, and scales as you grow. https://www.together.ai/products#inference, 2024.

[64] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 559–572.

[65] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 5998–6008.

[66] VERSTOEP, K., LANGENDOEN, K., AND BAL, H. E. Efficient reliable multicast on myrinet. In *Proceedings of the 1996 International Conference on Parallel Processing, ICCP 1996, Bloomingdale, IL, USA, August 12-16, 1996. Volume 3: Software* (1996), K. Pingali, Ed., IEEE Computer Society, pp. 156–165.

[67] WANG, A., CHANG, S., TIAN, H., WANG, H., YANG, H., LI, H., DU, R., AND CHENG, Y. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 443–457.

[68] WANG, J., HAN, J., WEI, X., SHEN, S., ZHANG, D., FANG, C., CHEN, R., YU, W., AND CHEN, H. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)* (July 2025), USENIX Association.

[69] WANG, K. A., HO, R., AND WU, P. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 39:1–39:16.

[70] WANG, W., GHOBADI, M., SHAKERI, K., ZHANG, Y., AND HASANI, N. Rail-only: A low-cost high-performance network for training llms with trillion parameters. In *IEEE Symposium on High-Performance Interconnects, HOTI 2024, Albuquerque, NM, USA, August 21-23, 2024* (2024), IEEE, pp. 1–10.

[71] WANG, Y., CHEN, Y., LI, Z., KANG, X., TANG, Z., HE, X., GUO, R., WANG, X., WANG, Q., ZHOU, A. C., AND CHU, X. Burstgpt: A real-world workload dataset to optimize llm serving systems. https://arxiv.org/abs/2401.17644, 2024.

[72] WEI, X., CHENG, R., YANG, Y., CHEN, R., AND CHEN, H. Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 987–1004.

[73] WEI, X., LU, F., CHEN, R., AND CHEN, H. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 121–136.

[74] WEI, X., LU, F., WANG, T., GU, J., YANG, Y., CHEN, R., AND CHEN, H. No provisioned concurrency: Fast rdma-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association.

[75] WU, B., LIU, S., ZHONG, Y., SUN, P., LIU, X., AND JIN, X. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *CoRR abs/2404.09526* (2024).

[76] YANG, Y., ZHAO, L., LI, Y., ZHANG, H., LI, J., ZHAO, M., CHEN, X., AND LI, K. Infless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 768–781.

[77] YU, G., JEONG, J. S., KIM, G., KIM, S., AND CHUN, B. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), M. K. Aguilera and H. Weatherspoon, Eds., USENIX Association, pp. 521–538.

[78] YU, M., YANG, R., JIA, C., SU, Z., YAO, S., LAN, T., YANG, Y., CHENG, Y., WANG, W., WANG, A., AND CHEN, R. lambdascale: Enabling fast scaling for serverless large language model inference, 2025.

[79] ZENG, S., XIE, M., GAO, S., CHEN, Y., AND LU, Y. Medusa: Accelerating serverless LLM inference with materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025* (2025), L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds., ACM, pp. 653–668.

[80] ZHANG, H., TANG, Y., KHANDELWAL, A., AND STOICA, I. SHEPHERD: serving dnns in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023* (2023), M. Balakrishnan and M. Ghobadi, Eds., USENIX Association, pp. 787–808.

[81] ZHONG, Y., LIU, S., CHEN, J., HU, J., ZHU, Y., LIU, X., JIN, X., AND ZHANG, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 193–210.

Instance type	Accelerators	Local SSD BW/GPU	Remote SSD BW/GPU	Network BW/GPU	Has NVLink	Price
a2-ultragpu-8g [35]	8 x A100(80 GB)	2.58 Gbps	0.29 Gbps	12.5 Gbps	✓	40.44 USD/h
p4d.24xlarge[9]	8 x A100(40 GB)	2.31 Gbps	-	100 Gbps	✓	45.039 USD/h
ml.hpcpni2.28xlarge[27]	8 x A100(80 GB)	4 Gbps	-	100 Gbps	✗	48.23 USD/h
p4de.24xlarge[9]	8 x A100(80 GB)	2.31 Gbps	-	100 Gbps	✓	56.328 USD/h
a3-highgpu-8g[35]	8 x H100	6.09 Gbps	0.97 Gbps	100 Gbps	✓	88.25 USD/h
a3-megagpu-8g[35]	8 x H100	6.09 Gbps	0.97 Gbps	200 Gbps	✓	Unavailable
p5.48xlarge [9]	8 x H100	9.8 Gbps	-	400 Gbps	✓	Unavailable

Table 2: *A survey of* MAAS *hardware configurations from GPU vendors.*

A Appendix

A.1 Notable implementation details

Network library We implement a communication library that abstracts both NVLink and RDMA to holistically transfer parameters, similar to DeepEP [24]. During our implementation, we found establishing communication group between machines is slow (e.g., 100 ms) when using off-the-shelf group communicators (e.g., NCCL [49]), which significantly limit the effectiveness of network-based scaling. Fortunately, we found that our plan only requires P2P communication between each pair of nodes. Therefore, we pre-create a connection pool that supports full-mesh connections on each. While the compute network (RDMA) has potential scalability issue [43], it only occurs when transferring small payloads and can be addressed using advanced RDMA transport like DCT [73].

Native serving engine with CUDA context pool. Before execution, a CUDA context with loaded kernels (cuModule) must be created on GPU. Creating such a CUDA context takes about 500 ms, and is non-negligible in serving instance autoscaling. To mitigate such an overhead, BLITZSCALE preserves a small CUDA context pool with pre-loaded kernels and transfers parameters to GPU within one of the existing CUDA contexts, similar to an existing work [40]. Furthermore, BLITZSCALE is built using C++ and native CUDA APIs, eliminating the overhead of initializing PyTorch (e.g. `dlopen`).

Fault tolerance. When machine failures occur, we will autoscale new instances using our scaling mechanism. One problem is that cached parameters on the failed machine are lost, so we need to redistribute these parameters to other machines to maintain our global parameter pool invariant. For other components in the system like scheduler or monitor failures, we follow the same procedure in existing work for recovery [56, 29].

A.2 Hardware configurations for MAAS

Table 2 lists the hardware configurations backed by typical MAAS systems.





Bayesian Code Diffusion for Efficient Automatic Deep Learning Program Optimization

Isu Jeong Seulki Lee

Ulsan National Institute of Science and Technology (UNIST)

Abstract

We introduce Bayesian code diffusion, a new deep learning program optimization strategy devised to accelerate the auto-tuning process of deep learning compilers. Using the concepts of *prior* and *posterior* distributions in the Bayesian framework and reformulating them in the context of deep learning program optimization, the proposed approach efficiently searches for optimal program code in a significantly reduced search space through an iterative diffusion of program code. To further enhance the efficiency of program optimization, we propose pre-training and fine-tuning for the cost model, which improves both the model's predictive accuracy and training efficiency. We implement Bayesian code diffusion in Ansor and evaluate its performance on a wide range of deep learning models on both CPUs and GPUs. Existing approaches struggle to reliably generate high-performing deep learning programs, i.e. achieving low program execution latency, across various configurations, including diverse deep learning model architectures and hardware platforms (CPU and GPU). In contrast, Bayesian code diffusion reduces the end-to-end compilation (optimization) time required to generate the equivalent program execution latency in various configurations, i.e., achieving up to 3.31× optimization speedup. This substantial improvement demonstrates that Bayesian code diffusion performs efficient and principled deep learning program optimization across a wide range of deep learning models, operators, and hardware (CPU and GPU).

1 Introduction

High-performance deep learning (tensor) programs are essential for constructing efficient deep learning systems. To achieve this, deep learning compilers, such as TVM [6], Glow [34], nGraph [10] and XLA [23], have been developed to generate deep learning programs optimized for various hardware architectures, such as CPUs, GPUs, and others. Without relying on hardware vendor-specific libraries, e.g., BLAS [22], Intel MKL [19], Nvidia cuBLAS [29], and cuDNN [30], they perform program optimizations by exploring the search space constructed from pre-defined rules and patterns, such as templates [6]. The search space is navigated by tuning its associated parameters, minimizing manual effort.

However, these deep learning compilers often result in suboptimal program performance, resulting in increased program execution latency. To generate improved deep learning (tensor) programs, the concept of auto-tuning [7] has been proposed, where a set of parameters that define the program codes are automatically searched and optimized. Notably, Ansor [47] constructs an expanded search space for each subgraph of the deep learning model through sketch generation, followed by random initialization and evolutionary search [41]. Although such auto-tuning techniques can identify deep learning programs that achieve lower execution latencies, extensive program exploration in an expanded search space often leads to increased program optimization time.

The increased optimization time observed in auto-tuning approaches highlights three opportunities for improvement. First, each subgraph is assigned an independent search space with a distinct set of optimizations, despite potential overlaps between them, which may be redundant. These redundancies present opportunities for improvement through reuse or shared optimization across subgraphs. Second, many existing auto-tuning methods [7, 47] perform a large number of random searches to identify optimal program codes, which can be time-consuming and cause inefficiencies in program optimization, despite the opportunity to start from better initial points. Lastly, the cost model used to predict the expected performance of the generated program, i.e., program execution latency, is trained without a learning strategy, presenting an opportunity to improve both predictive accuracy and learning efficiency through more principled training strategies. Although various approaches [12, 14, 26] have been proposed to exploit these opportunities, each has limited effectiveness in fully resolving these issues and presents its own set of limitations, as summarized in Tab. 1.

In this paper, we introduce the concept of Bayesian code diffusion and demonstrate its implementation, which enhances

Table 1: The comparison of state-of-the-art auto-tuning methods for deep learning (tensor) program optimization relative to Ansor [47] shows that the proposed Bayesian code diffusion enhances all features.

	Sample Program Sharing	Search Strategy Enhancement	Cost Model Improvement	Online Learning	Applicable HW CPU	Applicable HW GPU
SelectiveTuning (2019) [3]	✓	✓	✗	✓	✗	✓
Ansor (2020) [47]	✗	✗	✗	✓	✓	✓
ETO (2021) [12]	✓	✓	✗	✓	✗	✓
DietCode (2022) [46]	✓	✗	✗	✓	✗	✓
TransferTuning (2022) [14]	✓	✓	✗	✗	✓	✗
OneShotTuner (2022) [35]	✗	✗	✓	✗	✓	✓
FamilySeer (2023) [26]	✗	✗	✓	✓	✓	✓
BayesianCodeDiffusion (ours)	✓	✓	✓	✓	✓	✓

the efficiency of auto-tuning for optimizing deep learning (tensor) programs. We reformulate the problem of deep learning program optimization based on the Bayesian framework [27] and present an efficient solution for addressing it. Specifically, we leverage the concept of *prior*, represented by well-optimized program code for a specific subgraph, and propagate its parameter to a *posterior* subgraph—another similar subgraph to be optimized—to facilitate an efficient search for optimal programs in the program parameter space. Inspired by the observation that deep learning models often share similar subgraphs in their architectures, we iteratively transform the *posterior* program within the highly-likely parameter spaces close to the *prior*'s parameter distribution through the process we term *code diffusion*. This process enables an efficient and principled optimization of program parameters, in contrast to existing auto-tuning methods [7, 47] applying random search in an expansive program parameter space.

We also propose a cost model learning strategy (i.e., pre-training and fine-tuning) that adjusts the optimization order of program subgraphs to improve the convergence and predictive accuracy of the cost model. Auto-tuning generates a large number of program candidates, but measuring their latency on hardware is costly. To reduce this overhead, a cost model is trained during program compilation (optimization) to estimate the program latency based on a small subset of hardware measurements, following an online learning approach. However, existing approaches, such as round-robin or gradient-based sampling [47], yield overly diverse data samples, hindering the cost model's ability to generalize. In contrast, the proposed pre-training and fine-tuning strategy guides the cost model through a more structured data sequence, improving both learning efficiency and prediction quality. Initially, the cost model is trained using a set of *prior* program codes across subgraph clusters in a random order —we refer to as *pre-training*— to enhance its generalization capacity by learning diverse data generated through hardware measurements of heterogeneous *prior* subgraphs. Then, the cost model is trained with specific and homogeneous data generated by each cluster of similar subgraphs one by one —we refer to as *fine-tuning*— to improve its predictions on each subgraph cluster, on top of the generalization ability obtained during pre-training.

We implement the proposed Bayesian code diffusion on Ansor [47], demonstrating that it can be seamlessly and easily integrated into existing auto-tuning frameworks with minimal modification and effort. We conduct extensive experiments across a wide range of deep learning models, such as ResNet [16] and BERT [11]. The experimental results show that Bayesian code diffusion accelerates the end-to-end compilation (optimization) time of deep learning models by up to 3.31× while producing programs with execution latencies up to 1.13× faster than those generated by state-of-the-art auto-tuning methods [47]. The reduced compilation time is achieved by effective program sharing, efficient and principled program search, and online optimization on both CPU and GPU, without compromising on program execution latency. In contrast, previous methods are often limited by inefficient program search capabilities [26, 47], or constrained hardware compatibility (CPU/GPU) [3, 12, 46], or limited support for operators [46]. Some also rely on pre-compiled programs [14] or offline-trained cost models [35].

2 Background and Opportunities

2.1 Deep Learning Compilers and Auto-Tuning

Deep Learning Compilers. To accelerate deep learning workloads on target hardware architectures, specialized hardware vendor libraries, such as BLAS [22], Intel MLK [19], Nvidia cuBLAS [29], and cuDNN [30], have been widely used to generate efficient deep learning programs. However, developing these hardware-dependent libraries requires significant time and resources, making it difficult to keep up with the rapid emergence of state-of-the-art deep learning models and new hardware platforms. Intermediate Representation (IR)-based deep learning compilers [6] try to tackle this problem by being designed to operate across various hardware architectures without being constrained by hardware dependencies, alleviating the burden of manual optimization. These deep learning compilers, such as TVM [6], Glow [34], nGraph [10] and XLA [23], take deep learning workloads as input and apply a range of optimizations at both high-level and low-level IR to generate efficient deep learning program code without relying on hardware vendor libraries.

Auto-Tuning. Auto-tuning [7, 47] is a technique that automates the hardware-specific optimizations of the low-level IR lowered from the high-level IR graph—e.g., Relay IR [33] in TVM and HLO IR [15] in XLA— of the deep learning model. In the backend of the compiler, it sketches for optimal program parameters, such as loop tile size, from a vast set of candidates to ensure that the final program code performs best on the target hardware. For example, to explore efficient code implementation for the given tensor operations, AutoTVM [7] builds the search space of the parameters based on manually defined templates which demand considerable time and effort in their design. Consequently, the search space becomes dependent on manually predefined templates, which restricts the search scope and range. To solve this problem, Ansor [47] automatically constructs the search space through program sampling and fine-tuning stages, finding optimal program code without being restricted by limited templates.

Fig. 1 illustrates the auto-tuning process of Ansor [47]. Ansor first constructs the search space by generating a set of optimization rules, named sketch, for each subgraph. It then selects a subgraph and performs random initialization for the parameters of optimization rules belonging to the sketch, followed by iterative fine-tuning to optimize the parameters of these rules and generate executable complete program code. Among the resulting candidates, only those predicted by the cost model to have low latency are retained and evaluated on the target hardware. The cost model is trained with these evaluation results from scratch in an online learning paradigm. This process is repeated until the time budget is exhausted.

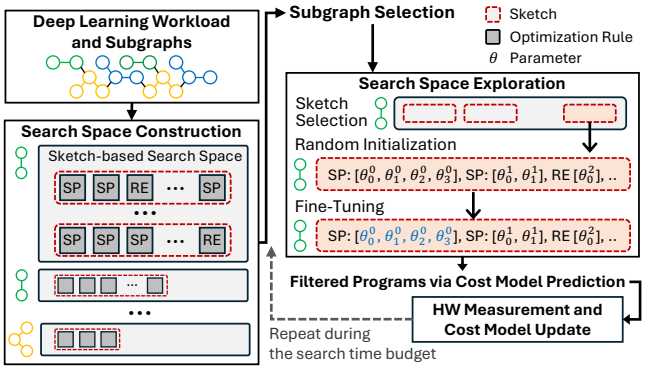


Figure 1: An overview of auto-tuning process in Ansor [47]. The details of the optimization rule, SP (Split-Step), and its parameters θ (integers) are described in Sec. 4.3.

2.2 Opportunities of Auto-Tuning

Although Ansor [47] automates finding optimal hardware-specific program code through program sampling and fine-tuning, it presents several opportunities, as described below.

Opportunity 1: Ignored Subgraph Similarity. In auto-tuning, the computation graph (i.e., high-level IR) is divided into subgraphs that can be optimized via pattern mapping or specific logic. Ansor [47] constructs the search space for each subgraph by recursively applying predefined optimization rules, such as reordering, multi-level tiling, and node fusion. Since this set of rules, named sketch, lacks specific parameters that define tile sizes and loop annotations, they must be determined by tuning (optimizing) parameters to generate executable programs. However, each parameter is determined independently through random initialization to produce executable program code. As a result, the optimal program parameters identified by one subgraph are not shared with others, even though there is potential for reuse when subgraphs exhibit similarities.

To reuse optimized program code, TransferTuning [14] adapts parameters previously optimized for a different deep learning model and transforms them to fit the recipient subgraph of the target deep learning model. However, this approach requires a pre-compiled deep learning program that closely resembles the recipient deep learning program, which limits its usability and applicability in practice. Although DietCode [46] creates a unified search space for microkernel operations and collectively optimizes them, it supports only a limited set of operators (e.g., dense and batch matmul), requiring the development of each individual operation and restricting implementations to only GPUs.

Opportunity 2: Randomized Fine-Tuning Initialization. To fine-tune program codes, Ansor [47] begins with a set of candidate complete programs, known as schedules, obtained through a combination of sketch generation and random parameter initilization. These initial programs are then iteratively fine-tuned with search algorithms, such as evolutionary operations [41], to produce new programs that can achieve improved performance, typically lower program execution latency. However, since these initial candidate programs are generated randomly, a large number of fine-tuning iterations is typically required to identify the optimal program code.

To reduce extensive random sampling and fine-tuning, SelectiveTuning [3] selects a subgraph in the high-level IR and optimizes its parameters through sufficient searches. The optimized parameters are then adapted to other subgraphs of similar structure through a one-time transformation based on predefined rules. While this approach can reduce random search iterations, relying on a one-shot single transformation without sufficient fine-tuning often results in suboptimal program performance, thus limiting its overall effectiveness.

Opportunity 3: Learning Strategy of Cost Models. Auto-tuners employ a cost model [47] that predicts the expected performance of the generated programs (i.e., execution latency) to select ones anticipated to perform better than others. The cost model is trained from scratch during compilation using hardware measurements in an online learning setting. Since optimization depends on the cost model, both the quality and quantity of the training data are critical for effective auto-tuning. In general, the training set is derived from programs executed using the round-robin and gradient with respect to program latency, without considering similarities, resulting in

the cost model being trained on overly broad and diverse data. If so, it must make predictions on a wide range of diverse programs, which may result in suboptimal prediction accuracy. Moreover, the diversity of data can prolong the training time, leading to unreliable predictions until the model convergence. Additionally, if there exist similar programs, they tend to produce similar data samples, causing the model to learn redundant and ineffective samples. This not only fails to enhance the model's learning performance but also wastes time and resources [26, 32, 42].

To accelerate convergence and improve prediction performance of the cost model, FamilySeer [26] groups similar subgraphs and trains a separate cost model for each group. Although this approach can reduce program search time by using a dedicated cost model for each group, the limited range and diversity of training datasets can often result in suboptimal programs, as each group is likely to generate only similar program samples. Moreover, it does not reuse optimized parameters across subgraphs, resulting in inefficient search.

3 Methodological Idea

To take advantage of the aforementioned opportunities of auto-tuning in deep learning compilers, we propose Bayesian code diffusion, of which the key idea is illustrated in Fig. 2.

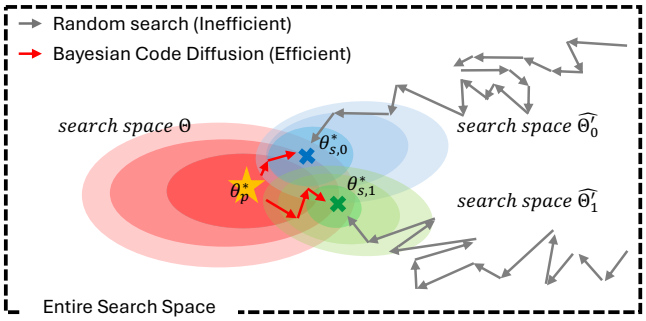


Figure 2: The concept of Bayesian code diffusion: a sufficiently optimized *prior* parameter of one subgraph (θ^*_p) is propagated to similar subgraphs (*posteriors*). Then, *posterior* parameters ($\theta^*_{s,\{0,1\}}$) are derived and refined from the *prior* for each subgraph via code diffusion using a Bayesian formulation, enabling efficient deep learning program optimization.

It first selects a *prior* subgraph and performs a sufficient search to find its optimal program parameters. These optimized parameters are then propagated to structurally similar subgraphs, referred to as *posterior* subgraphs, from a Bayesian perspective. The initial search space of each *posterior* subgraph is restricted to that of the *prior* subgraph, and optimization is performed in the vicinity of the *prior*'s optimal parameters through *prior* propagation. Then these parameters are fine-tuned by adding some noise to find the optimal configuration for the *posterior* subgraph. This process is referred to as *code diffusion*. Repeating this code diffusion process enables the effective propagation of the most promising area of the search space among similar subgraphs, effectively reducing fine-tuning iterations. Also, in this process, the cost model undergoes pre-training using *prior* subgraphs and is subsequently fine-tuned on *posterior* subgraphs, allowing it to learn efficiently and effectively. The following subsection presents the hypothetical formulation of Bayesian code diffusion (Sec. 3.1) and empirical findings that support the idea and validity (Sec. 3.2).

3.1 Theoretical Formulation

Prior Parameters. Bayesian code diffusion first begins by clustering subgraphs of the deep learning model according to their similarities and selecting one of them as the *prior* subgraph $\mathcal{G}_p$ for each cluster. Given $\mathcal{G}_p$, the optimization task is to find the optimal parameter θ^*_p generating optimal program code for $\mathcal{G}_p$. If we employ the Bayesian formulation to this, with an abuse of notations, the optimization task can be expressed as maximizing the following conditional likelihood.

$$\theta^*_p = \underset{\theta_p}{\operatorname{argmax}} f_{\min}(c(\theta_p) \mid \theta_p) \tag{1}$$

Here, $c(\theta)$ is the program code defined by the parameter θ, and $f_{\min}(c(\theta))$ is defined as the hypothetical probability density regarding whether the execution latency of $c(\theta)$ is the minimal or not among all $\theta \in \Theta$ explored so far, as follows:

$$f_{\min}(c(\theta)) = \begin{cases} 1 & \text{if } l(c(\theta)) < l(c(\theta')) \ \forall\ \theta' \in \Theta \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $l(c(\theta))$ is the execution latency of $c(\theta)$, and Θ is the total set of schedules explored so far. Now assume a subgraph $\mathcal{G}_s$, presumed to be similar to $\mathcal{G}_p$, is given, and the task is to find its optimal parameter θ^*_s. In principle, the optimal program code $c(\theta^*_s)$, which yields the minimum latency for $\mathcal{G}_s$, can be found if all possible parameters $\theta \in \hat{\Theta}$ are explored. This can be expressed as a hypothetical marginal distribution over all possible $\theta \in \hat{\Theta}$, which is given by:

$$f_{\min}(c(\theta^*_s)) = \sum_{\theta \in \hat{\Theta}} f_{\min}(c(\theta) \mid \theta) f(\theta) \tag{3}$$

In Ansor [47], the search space $\hat{\Theta}$ is defined by a sketch—a sequence of optimization rules—and consists of all possible parameter combinations for those rules, defined as $\hat{\Theta} = \prod_{i=0}^{K} \theta_i$, where K is the number of rules belonging in the sketch. However, due to the vastness of $\hat{\Theta}$, exhaustively exploring all configurations $\theta \in \hat{\Theta}$ is infeasible, necessitating the derivation of a feasible yet effective subspace $\hat{\Theta}' \subseteq \hat{\Theta}$.

Prior Propagation. To reduce $\hat{\Theta}$ into a feasible yet effective search space $\hat{\Theta}'$, we take θ^*_p, optimized for the *prior* subgraph $\mathcal{G}_p$ in Eq. (1), as the *prior* parameter for $\mathcal{G}_s$ by setting $\hat{\Theta}' = \{\theta^*_p\}$. Specifically, we may regard θ^*_p as the mode of the hypothetical Gaussian distribution for θ_p, i.e.,

$f(\theta_p) = \mathcal{N}(\theta_p^*, \sigma_p^2)$ with the mean θ_p^* and variance σ_p^2. Based on our observations that θ_p^* and θ_s^* are likely to be close in the search space, i.e., $\theta_p^* \simeq \theta_s^*$ as $\mathcal{G}_p \simeq \mathcal{G}_s$, we also assume the hypothetical distribution of θ_s is also likely to be close to that of θ_p, and thus take $f(\theta_p) = \mathcal{N}(\theta_p^*, \sigma_p^2)$ as the hypothetical *prior* distribution of θ_s as $f(\theta_s) = \mathcal{N}(\theta_p^*, \sigma_s^2)$. From this, we can avoid computing the intractable marginal in Eq. (3) over all possible $\theta \in \hat{\Theta}$, but instead evaluate the following hypothetical *posterior* as a joint distribution:

$$\theta_s^* = \underset{\theta_s \in \hat{\Theta}'}{\operatorname{argmax}}\, f_{\min}(c(\theta_s), \theta_s) = \underset{\theta_s \in \hat{\Theta}'}{\operatorname{argmax}}\, f_{\min}(c(\theta_s) \mid \theta_s) f(\theta_s) \quad (4)$$

where $\hat{\Theta}' = \{\theta_p^*\}$ for the beginning. To identify θ_s^*, we evaluate the hypothetical maximum joint probability in Eq. (4), as $f_{\min}(c(\theta_s^*))$ always returns one by definition as in Eq. (2), and it does not provide the confidence that θ_s^* found so far is indeed the true optimum. However, when multiplied with $f(\theta_s)$, $f_{\min}(c(\theta_s) \mid \theta_s) f(\theta_s)$ can quantify the confidence that the found θ_s^* is the true optimal parameters, i.e., the lower the value, the more likely that θ_s^* is the true optimum.

Code (Parameter) Diffusion. After the *prior* propagation, the search space for $\mathcal{G}_s$ is defined as $\hat{\Theta}' = \{\theta_p^*\}$, where θ_p^* is considered the initial candidate for the optimal parameter for $\mathcal{G}_s$, i.e., $\theta_s^{(0)} = \theta_p^*$, and Eq. (4) is evaluated with it. to find an improved θ_s^*, we apply code diffusion to θ_p^*, which hypothetically generates the next *posterior* candidate $\theta_s^{(1)}$ from θ_p^*, and subsequently the program code $c(\theta_s^{(1)})$. At the tth optimization iteration, the *posterior* parameter $\theta_s^{(t)}$ is obtained by hypothetically diffusing θ_p^* as follows:

$$\theta_s^{(t)} = \sqrt{1 - \sigma_{s,t}^2}\,\theta_p^* + \sqrt{\sigma_{s,t}^2}\,\varepsilon \quad (5)$$

where $\varepsilon \sim \mathcal{N}(0, \mathbf{I})$, and $\sigma_{s,t}^2 < 1$ is the variance of the diffusion for $\theta_s^{(t)}$. The newly diffused parameter $\theta_s^{(t)}$ is then appended to the search space as $\hat{\Theta}' = \{\theta_s^{(0)}, \theta_s^{(1)}, \ldots, \theta_s^{(t-1)}\} \cup \{\theta_s^{(t)}\}$ and evaluated using Eq. (4). If $\theta_s^{(t)}$ yields the maximum value for Eq. (4), it is deemed the optimal, i.e., $\theta_s^* = \theta_s^{(t)}$. By repeating code diffusion, $\theta_s^{(t)}$ gradually difusses from θ_p^*, expanding the search space from the *prior* distribution $f(\theta_p)$, in contrast to many existing approaches [3, 14] that operate within a fixed search space. By searching for θ_s^* starting from the mode of the *prior* distribution θ_p^* and iteratively diffusing θ_s around it, θ_s^* can be efficiently found within highly likely search spaces.

3.2 Empirical Support

The concept Bayesian code diffusion is based on the following empirical findings, which form the basis of its core principles.

Commonalities in Incomplete Programs (Sketches). Table 2 presents a list of subgraphs of BERT [11], indicating that five out of eight subgraphs share the same incomplete programs, i.e., the sets of optimization rules (sketches) S_0 based on the similarity of their operations when utilizing Ansor [47]. Nevertheless, even though some subgraphs exhibit identical forms of sketches, independent program optimization tasks and distinct search spaces are assigned for each subgraph. This arises from potential differences in their details, e.g., tensor dimensions, resulting in a redundant and inefficient search process. Given that many deep learning models exhibit repetitive operations in their network architectures, there exists a potential for the proposed *prior* propagation, contingent upon the subgraphs sharing the same sketches from which program code parameters can be effectively diffused.

Subgraph	Subgraph Description	Sketches
$\mathcal{G}_0$	batch_matmul_3	S_0
$\mathcal{G}_1$	variance	S_1
$\mathcal{G}_2$	batch_matmul_1	S_0
$\mathcal{G}_3$	softmax_broadcast_to	S_2
$\mathcal{G}_4$	batch_matmul	S_0
$\mathcal{G}_5$	batch_matmul_4	S_0
$\mathcal{G}_6$	batch_matmul_2	S_0
$\mathcal{G}_7$	mean	S_3

Table 2: (Left) The subgraphs in BERT [11] and their sketches obtained by applying Ansor [47], where subgraphs $\mathcal{G}_{0,2,4,5,6}$ produce the same sketches $S_0 = \{s_0^0, s_1^0, s_2^0\}$.

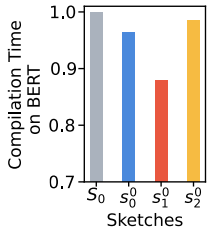


Figure 3: (Right) The compilation time of the subgraphs $\mathcal{G}_{0,2,4,5,6}$ taken by Ansor [47] on the operation `batch_matmul` in BERT [11] using the sketches S_0 and each sketch $s_i^0 \in S_0$ on a GPU. Here, '1.0 (gray)' denotes the time required when using all sketches $S_0 = \{s_0^0, s_1^0, s_2^0\}$.

Sketch-based Optimization. Figure 3 illustrates an example in which a set of subgraphs deriving the same sketches $S_0 = \{s_0^0, s_1^0, s_2^0\}$, i.e., $\{\mathcal{G}_0, \mathcal{G}_2, \mathcal{G}_4, \mathcal{G}_5, \mathcal{G}_6\}$, can be effectively optimized from the single sketch, s_1^0. The bars in the figure present the optimization (compilation) time required to achieve the same program latency when using only a single sketch ($s_i^0 \in S_0$), compared to using all sketches (S_0). It is observable that rather than exhaustively exploring the parameter search space of all sketches, it is more efficient to focus on a single sketch. Thus, if one of the subgraphs sharing the same sketches identifies the optimal parameters for one of them, the others can also derive optimized parameters from the same sketch with reduced search time. This indicates that the parameter search space can be effectively reduced by optimizing a common single sketch among similar subgraphs.

Parameter Distances. Fig. 4 (left) shows the distances between the optimized configuration, i.e., sketches and their parameters, for the subgraphs that have the same incomplete programs (sketches). It is compared with the distances between the configuration for subgraphs that have different sketches (right). The figure shows that the distances within the identical sketches cluster are significantly closer compared to those across the sketches cluster. This suggests that it is more likely to identify the optimal configuration by starting from one of the optimal parameters for the same sketch and taking it as a *prior* to explore in its vicinity through code diffusion.

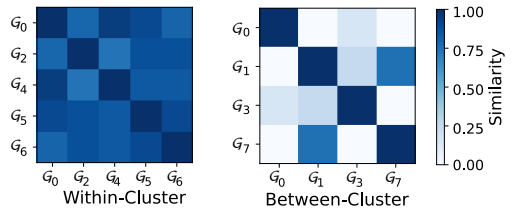


Figure 4: The cosine similarities between configurations optimized by Ansor [47]. The subgraphs $\{\mathcal{G}_0, \mathcal{G}_2, \mathcal{G}_4, \mathcal{G}_5, \mathcal{G}_6\}$ are optimized from the same sketch, while $\{\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_3, \mathcal{G}_7\}$ have different sketches. The darker, the higher similarity. The details of distance measurement is in Fig. 10.

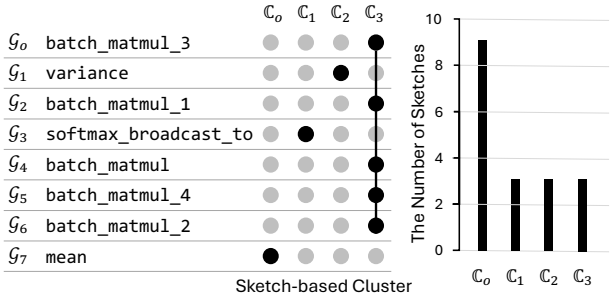


Figure 5: An example of sketch clustering applied to BERT [11], comprising four clusters: $\mathbb{C}_0$, $\mathbb{C}_1$, $\mathbb{C}_2$, and $\mathbb{C}_3$. Subgraphs within the same cluster are linked by vertical lines. As shown in Fig. 3, subgraphs with identical *sketches* are grouped into the same cluster. The figure on the right shows the number of *sketches* for each cluster. For example, $\mathbb{C}_3$ consists of subgraphs that have *sketches* $S_0 = \{s_0^0, s_0^1, s_0^2\}$.

4 Implementation

We describe how Bayesian code diffusion (Sec. 3.1) is implemented within deep learning compilers, based on Ansor [47]. The overall procedure of Bayesian code diffusion is summarized in Alg. 1, and an example is illustrated in Fig. 6.

4.1 Sketch-based Subgraph Clustering

The first step of Bayesian code diffusion is to cluster a set of subgraphs into distinct groups. Existing approaches, such as Nimble [21], SelectiveTuning [3], and FamilySeer [26], group subgraphs based on their operations, which do not necessarily produce identical sketches due to variations such as tensor dimensions. Since different sketches correspond to distinct optimization spaces and configurations, the optimal program code may vary accordingly, thus increasing the search space to explore. For this reason, we cluster subgraphs based on their sketches to enhance both the likelihood of discovering effective prior program parameter θ_p^* and the efficiency of code diffusion, motivated by the observations provided in Sec. 3.2. The search space of *posterior* subgraphs is initialized to the parameter search space for a sketch where θ_p^* is observed, thereby reducing the overall search space efficiently. As shown in Fig. 3, sketch-based optimization not only simplifies the optimization scopes and processes but also reduces the search space, as the proposed code diffusion occurs solely within the parameters, without involving optimization rules. This approach enables parameters to evolve efficiently during code diffusion, facilitating the automatic discovery of optimal parameters. Fig. 5 shows an example of sketch-based subgraph clustering. Nonetheless, Bayesian code diffusion can also be applied to operation-based clustering [3].

Algorithm 1 Bayesian code diffusion. $\theta^{*,i}$ denotes the optimal parameter for subgraph $\mathcal{G}^i$, and $\mathcal{B}$ indicates the time budget. Hyper-parameters ε_e and ε_m define the number of iterations for fine-tuning and hardware measurement, respectively.

Input: Subgraphs $G = [\mathcal{G}^0, \cdots, \mathcal{G}^n]$
Output: Optimal parameters θ^* for G
BayesianCodeDiffusion begin
  $[G^1, \cdots, G^m] \leftarrow$ `SketchBasedClustering`(G)
  for *each cluster* $G^i \in [G^1, \cdots, G^m]$ **do**
    $G_{\text{prior}} \leftarrow G_{\text{prior}} \cup \{\mathcal{G}^i_{\text{prior}} \mid \mathcal{G}^i_{\text{prior}} \in G^i\}$ [Eq. 6, 7]
    $G_{\text{posterior}} \leftarrow G_{\text{posterior}} \cup \{\mathcal{G}^{i,j} \in G^i \mid \mathcal{G}^{i,j} \neq \mathcal{G}^i_{\text{prior}}\}$
    for $\mathcal{G}^i_{prior} \in G_{prior}$ **do**
      `PriorOptimization`$(\mathcal{G}^i_{\text{prior}}, \varepsilon_e, \varepsilon_m)$
    end
    for $\mathcal{G}^{i,j} \in G_{posterior}$ **do**
      `CodeDiffusion`$(\mathcal{G}^{i,j}, \theta^{*,i}, \frac{\varepsilon_e}{2}, \frac{\varepsilon_m}{2})$
    end
  end
  while $\mathcal{B} > 0$ **do**
    $\mathcal{G}^{i,j} \leftarrow$ `GradientBasedSelection`(G)
    `CodeDiffusion`$(\mathcal{G}^{i,j}, \theta^{*,i}, \frac{\varepsilon_e}{2}, \frac{\varepsilon_m}{2})$
  end
end
`PriorOptimization`$(\mathcal{G}, \varepsilon_e, \varepsilon_m)$ **begin**
  $S \leftarrow$ `SketchGeneration`$(\mathcal{G})$
  `RandomInitialization`$(\mathcal{G})$
  `FineTuning`$(\mathcal{G}, \varepsilon_e)$
  $\mathcal{B} \leftarrow \mathcal{B} - \varepsilon_m$
end
`CodeDiffusion`$(\mathcal{G}, \theta^*, \varepsilon_e, \varepsilon_m)$ **begin**
  $S \leftarrow$ `SketchGeneration`$(\mathcal{G})$
  $S \leftarrow \{s \in S \mid s$ is *sketch* for $\theta^*\}$
  `PriorPropagation`$(\mathcal{G})$ [Fig. 8, Eq. 8, 9]
  `FineTuning`$(\mathcal{G}, \varepsilon_e)$
  $\mathcal{B} \leftarrow \mathcal{B} - \varepsilon_m$
end

4.2 Prior Selection and Optimization

The second step of Bayesian code diffusion is to select a *prior* subgraph $\mathcal{G}_p$ in each cluster and optimize it. Since subgraphs in the same cluster share structurally identical operations but differ in their tensor dimensions, we evaluate their similarity based on these tensor dimensions among them. For example, the subgraphs $\mathcal{G}_{0,2,4,5,6}$ in Fig. 3 all perform the same operation, `batch_matmul`, yet exhibit variations in their tensor dimensions. Then, the *prior* subgraph $\mathcal{G}_p$ is selected such that its tensor dimensions are closest to those of all the other

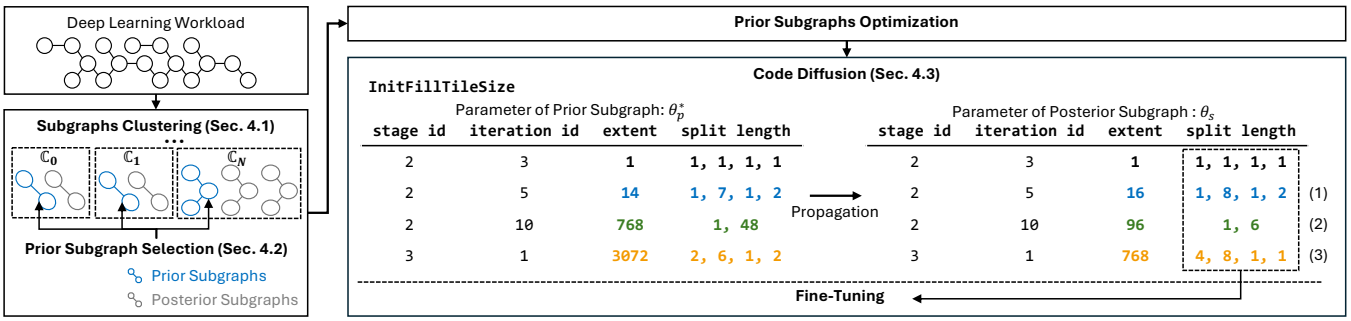


Figure 6: Given a set of subgraphs from a deep learning model, they are first clustered according to their sketches derived by Ansor [47]. Within each subgraph cluster, a prior subgraph is selected and optimized. Subsequently, the remaining subgraphs are optimized through prior parameter propagation, followed by iterative code diffusion using the Bayesian implementation.

Table 3: The examples of parameter initialization rules for program sampling used in Ansor [47] for both CPU and GPU.

CPU	GPU
InitFillTileSize	InitFillTileSize
InitUnroll	InitUnroll
InitChangeComputeLocation	
InitVectorization	
InitParallel	

subgraphs in the cluster while maintaining distinction from the subgraphs in different subgraph clusters. It is based on the assumption that the subgraph with tensor dimensions most similar to those of all other subgraphs is expected to enhance the effectiveness of code diffusion for the *posterior* subgraphs (Sec. 3.2), which also improves the code compatibility.

Given a set of N subgraphs $G = \{\mathcal{G}^0, \mathcal{G}^1, \cdots, \mathcal{G}^N\}$ in a subgraph cluster, each subgraph $\mathcal{G}^n$ consists of an operator sequence $o^n = [o_0^n, o_1^n, \cdots, o_K^n]$ with their tensor dimensions $[\mathbf{i}_0^n, \mathbf{i}_1^n, \cdots, \mathbf{i}_J^n]$. To compose the tensor dimension vector $\mathbf{i}^n$ for o^n, $[\mathbf{i}_0^n, \mathbf{i}_1^n, \cdots, \mathbf{i}_J^n]$ is concatenated as $\mathbf{i}^n = \text{concat}(\mathbf{i}_0^n, \mathbf{i}_1^n, \cdots, \mathbf{i}_J^n)$. Then, cosine similarities between all pairs of $\mathbf{i}^p$ and $\mathbf{i}^q$ are computed to construct the similarity matrix $\mathbf{S} \in \mathbb{R}^{N \times N}$, where the (p,q)th element $s_{p,q}$ is defined as:

$$s_{p,q} = [\mathbf{i}^p \cdot \mathbf{i}^q \;/\; \|\mathbf{i}^p\| \cdot \|\mathbf{i}^q\|] \;\; \forall \;\; 1 \le p,q \le N. \tag{6}$$

Given the similarity matrix $\mathbf{S}$, the *prior* subgraph $\mathcal{G}_p$ is chosen to maximize the average similarity of tensor dimensions across the elements of each row of $\mathbf{S}$, which is given by:

$$\underset{1 \le p \le N}{\text{argmax}} \frac{1}{N} \sum_{q=1}^{N} s_{p,q} \;\; \forall \;\; s_{p,q} \in \mathbf{S} \tag{7}$$

For example, consider three simple subgraphs, $\{\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2\}$. The operation in $\mathcal{G}_0$ is $[[1,1]] + [[2,2]]$, whereas in $\mathcal{G}_1$, it is $[[1,1],[2,2]] + [[3,3],[4,4]]$. Additionally, the operation in $\mathcal{G}_2$ is defined as $[[1,1,1],[2,2,2]] + [[3,3,3],[4,4,4]]$. Although these subgraphs are structurally identical, i.e., execute same operation, `add`, they differ in their tensor dimensions, i.e., the tensor in $\mathcal{G}_0$ has dimension $(1,2,1,2)$, in $\mathcal{G}_1$ dimension $(2,2,2,2)$, and in $\mathcal{G}_2$ dimension $(2,3,2,3)$. Given tensor dimension similarities defined as $Sim(\mathcal{G}_0, \mathcal{G}_1) = 0.95, Sim(\mathcal{G}_1, \mathcal{G}_2) = 0.98, Sim(\mathcal{G}_2, \mathcal{G}_3) = 0.99$. We select $\mathcal{G}_2$ as the *prior* subgraph due to its higher overall similarity to other subgraphs within the cluster.

In this way, $\mathcal{G}_p$ becomes the subgraph exhibiting the highest similarity with the other subgraphs in terms of tensor dimensions. Once $\mathcal{G}_p$ is chosen, the optimal parameter θ_p^* is explored within the search space corresponding to $\mathcal{G}_p$ by evaluating Eq. (1) through random search, with θ_p sampled from a uniform distribution. This exploration is repeated in proportion to the size of the cluster, i.e., the number of subgraphs $|\mathcal{G}|$. During this process, the cost model is trained using diverse data samples generated by various *prior* subgraphs with differing configurations within each cluster, improving its potential generalization ability across clusters. We refer to this training phase as pre-training.

4.3 Code Diffusion for Posterior Optimization

The final step of Bayesian code diffusion is to propagate the *prior* parameters to the other subgraphs in the same cluster and apply code diffusion by iteratively searching for the optimal *posterior* parameters for each subgraph alongside the *prior* parameter distribution. In auto-tuning frameworks like Ansor [47], which comprises random parameter initialization and fine-tuning, code diffusion is applied to random parameter initialization, subsequently facilitating the efficient fine-tuning of sampled programs. Given a sketch, the parameter initialization rules, shown in Tab. 3, are applied for program sampling, performing optimization tasks such as tiling, parallelization, and vectorization. In existing methods [47], the parameters governing these rules are typically searched randomly, causing inefficiency in optimization, which can be accelerated by diffusing the *prior* parameter. Here, we describe how code diffusion is applied to `InitFillTileSize` in Ansor [47], which works on both CPU and GPU. The implementations for other rules are similar and thus omitted. The role of `InitFillTileSize` is to initialize the parameters of the split-step optimization rule (SP), which divides a loop (iterator) into two or more nested loops. It assigns tile sizes

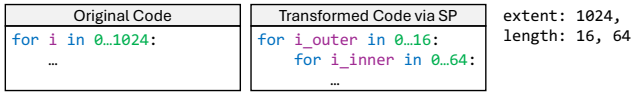


Figure 7: An example code transformation using the split-step (SP) with `extent=1024` and `length=[16,64]`.

based on two parameters, i.e., `extent` and `length`, to fit the extent of each loop axis, as illustrated in Fig. 7. The `extent` refers to the length of the axis to split and `length` denotes the multiple split factors. Given `extent`, code diffusion searches for the optimal parameters for `length` by diffusing the *prior* parameter θ_p^* to the *posterior* parameter θ_s, as shown in Fig. 6.

Although the optimization task for each subgraph executes the split-step at the same stage and iteration, the `length` of the loops to be split differ among subgraphs, which complicates the application of the *prior* parameters. To enable code diffusion across loops of varying `length`, we devise three mechanisms of code diffusion, which implement Eq. (5) in Sec. 3.1 without knowing the exact true probability distribution $f(\theta_s) = \mathcal{N}(\theta_p^*, \sigma_s^2)$.

First, if the `extent` of the split-step of the *prior* subgraph $\mathcal{G}_p$ matches that of the *posterior* subgraph $\mathcal{G}_s$, the same `length` of θ_p^* is applied to θ_s. Conversely, if the `extent` differ, the `length` in the *prior* parameter θ_p^* is diffused to θ_s such that θ_s is closest to the `length` of θ_p^* among all `length` derivable from the `extent` of θ_s. In particular, by using the rule that `length` consists of a set of divisors of the `extent`, we choose the divisor of the `extent` of θ_s closest to `length` of θ_p^*. Let e_p and e_s be the `extent` of $\mathcal{G}_p$ and $\mathcal{G}_s$, respectively, and the `length` of e_p is given as $\mathbf{l_s} = \{l_p^0, \cdots, l_p^n\}$. Then, the `length` $\bar{\mathbf{l}}_s$ of the *posterior* parameter θ_s is diffused as:

$$\bar{\mathbf{l}}_s = \{\bar{l}_s^i \in \mathbf{l}_s \mid \forall l_p^j \in \mathbf{l}_p, l_s^i = \min|l_p^j - l_s^i|\} \quad \text{where } \mathbf{l}_s = \{l_s^i \in \mathbb{Z}^+ \mid l_s^i \text{ divides } e_s\}. \tag{8}$$

For example, the `length` $\mathbf{l}_p$ of θ_p^*:$[1,7,1,2]$ is diffused to the $\bar{\mathbf{l}}_s$ of θ_s:$[1,8,1,2]$, as shown (1) in Fig. 6. The blue box in Fig. 8 shows the code implementation for this case.

Second, the `length` $\bar{\mathbf{l}}_s$ of θ_s is diffused based on the ratio of the `extent` and `length` of θ_p^*, as follows:

$$\bar{\mathbf{l}}_s = \{\bar{l}_s^i \in \mathbf{l}_s \mid l_s^i = \min|l_s^i - \hat{l}_s^i|, \hat{l}_s^i = \left\lfloor l_p^i \cdot \frac{e_s}{e_p} \right\rfloor \forall\, l_p^i \in \mathbf{l}_p\} \tag{9}$$

which corresponds to the green box in Fig. 8. For instance, $\mathbf{l}_p$ of θ_p^*:$[1,48]$ is diffused to θ_s:$[1,6]$, as shown (2) in Fig. 6. This ensures that the `length` of θ_s is proportionally aligned with that of θ_p^*, thus diffusing them in close proximity to the hypothetical *prior* parameter distribution $f(\theta_p)$.

Lastly, to ensure diversity in search spaces, the `length` of θ_s is generated randomly, without code diffusion, as shown in the yellow box in Fig. 8 with an example given (3) in Fig. 6. Given that θ_s^* can be possibly far from θ_p^*, this diversification process should be considered in optimization.

```
InitFillTileSize::Apply(search_policy, state){
  _extent = prior ->extent;
  _legnths = prior ->lengths;

  if (_extent == extent){ lengths = _lengths}
  else {
    random_choice = random_device();
    if (random_choice == 0){
      if (extent % mul(_lengths) == 0){
        lengths = _lengths;
      } else {
        divisors_of_extent = calculate_divisors(_extent);
        lengths = map_lengths_to_cloest_divisors(_lengths);
      }          Reuse Prior θ*p or diffuse θ*p to θs (closest to θ*p)
    }

    else if (random_choice == 1){
      divisors_of_extent = calculate_divisors(_extent);
      lengths = map_lengths_to_cloest_divisors(
      _lengths*(_extent/extent));
    }      Reuse Prior θ*p or diffuse θ*p to θs (based on the extent)

    else if (random_choice == 2){
      /*** Randomly Select for Bias ***/
    }                                   Random Selection for Bias
}}
```

Figure 8: The implementation of for `InitFillTileSize` in Ansor [47], showing three cases of code diffusion for posterior optimization (i.e., blue, green, and yellow boxes).

Among these three code diffusion methods, one of them is randomly chosen for the *posterior* subgraph optimization at every t-th search iteration. From this, the *posterior* parameter is diffused to $\{\bar{\mathbf{l}}_s\} \in \theta_s^{(t)}$ and then appended to the search space as $\hat{\Theta}' = \{\theta_s^{(0)}, \theta_s^{(1)}, \ldots, \theta_s^{(t-1)}\} \cup \{\theta_s^{(t)}\}$. In this way, $f(\theta_s) = \mathcal{N}(\theta_p^*, \sigma_s^2)$ can be effectively implemented, allowing for identifying θ_s^* that maximizes Eq. (4) and quantifying the confidence whether θ_s^* is the true optimal parameter or not, i.e., the lower the value of $f_{\min}(p(\theta_s) \mid \theta_s) f(\theta_s)$, the higher it is likely to be true optimal. Fig. 9 shows an example of the *posterior* program $c(\theta_s^*)$ optimized from the *prior* program $c(\theta_p^*)$ through code diffusion.

```
Placeholder: p0, p1
parallel b.0@i.0@j.0@ (0,16)
  T_batch_matmul_NN.local auto_unroll: 64
  for i_c.1 (0,7)
    for k.0 (0,768)
      for j_c.2 (0,3)
        for k.1 (0,4)
          for j_c.3 (0,32)
            T_batch_matmul_NN.local = ...
  for i.1 (0,7)
    for j.1 (0,96)
      T_batch_matmul_NN = ...
```

$c(\theta_p^*)$

```
Placeholder: p0, p1
parallel b.0@i.0@j.0@ (0,48)
  T_batch_matmul_NN.local auto_unroll: 16
  for i_c.1 (0,7)
    for k.0 (0,192)
      for j_c.2 (0,2)
        for k.1 (0,4)
          vectorize j_c.3 (0,16)
            T_batch_matmul_NN.local = ...
  for i.1 (0,7)
    vectorize j.1 (0,32)
      T_batch_matmul_NN = ...
```

$c(\theta_s^*)$

Figure 9: An example of the posterior program code $c(\theta_s^*)$ (right) diffused from the prior program code $c(\theta_p^*)$ (left).

The true distribution $f(\theta_s)$ described in Sec. 3.1 is challenging to obtain directly within the program parameter space. However, it can be estimated by evaluating the distance between the mode (mean) of the *prior* distribution, θ_p^*, and the *prior* parameter θ_s. Fig. 10 illustrates the schedule encoding process used for distance measurement. The schedule comprises optimization rules (represented as text) and their corresponding parameters (represented as a list of numbers) [47]. Due to the combination of text and parameters existing within a dynamic range—caused by differences in the type and number of optimization rules across sketches—directly comparing

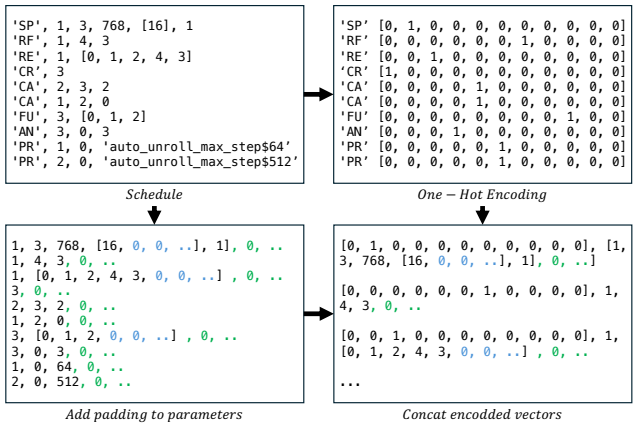


Figure 10: The process of converting a schedule [47] into a vector representation for measuring distances between schedules: the blue and green text indicate padding applied to standardize the vector's shape.

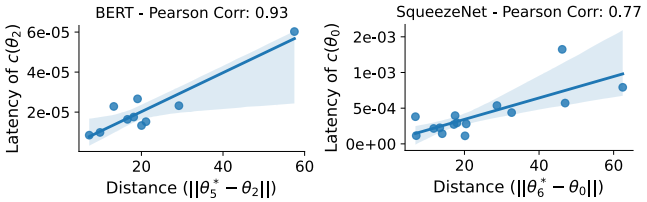


Figure 11: The latencies (ms) of the generated programs measured on CPU over the norm distance between optimal parameter θ^* and the parameter θ of another subgraph in Ansor [47] for BERT [11] (left) and SqueezeNet [18] (right).

distances or similarities between schedules becomes complex. Consequently, encoding these schedules as vectors is necessary. Here, a total of 11 optimization rules are employed, including SP (split-step), FU (fuse), among others [47]. The textual representations of these rules are converted into a one-hot vector of length 11. Subsequently, the corresponding parameters are zero-padded to transform the varying lists of numbers into a static length for uniformity. In Fig. 10, blue text denotes zero padding applied to specific parameters within identical rules, while green text represents zero padding applied to parameters across different rules. Finally, the one-hot and padded vectors are concatenated to form a unified representation. Figure 11 shows the correlation between the distance between the optimal parameter θ_a^* and θ_b of the other subgraph and the latency. This figure demonstrates that the program latency $l(c(\theta_b))$ decreases as the distance between θ_b and θ_a^* diminishes, indicating that a more optimal program can be discovered when θ_b is diffused from θ_a^*.

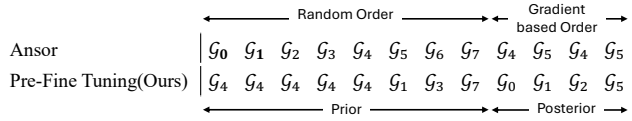


Figure 12: The optimization sequence of BERT [11] subgraphs of Ansor [47] and Bayesian code diffusion (Pre-Fine Tuning), which trains the cost model with datasets generated in different orders, with subgraphs optimized by Ansor.

5 Cost Model of Bayesian Code Diffusion

In auto-tuning, the cost model is essential for evaluating candidate programs by filtering out inefficient ones. Surviving candidates are executed on hardware, generating execution results used to refine the model's prediction accuracy. By continuously learning from online execution results, the cost model gradually improves its predictive performance from scratch within the online learning paradigm With an accurate cost model, program execution can be selectively guided, substantially reducing the need for exhaustive hardware measurements and accelerating the compilation time. For example, Tenset [48] has demonstrated that a cost model pre-trained on a large-scale dataset can accelerate search time.

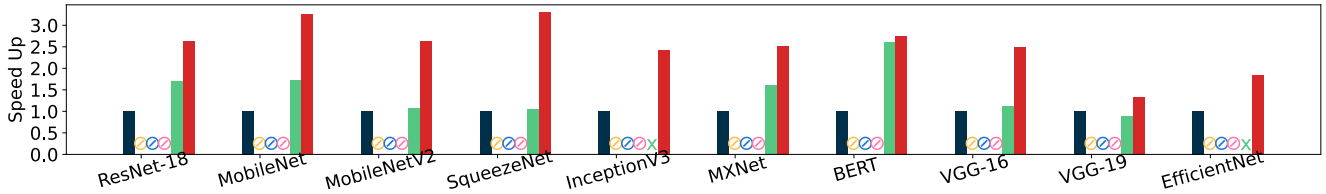
Two studies highlight the importance of the cost model from different perspectives. When the training data for the cost model is biased by the search method, the stability of its predictive accuracy can deteriorate [28], while FamilySeer [26] deliberately biases the data by grouping subgraphs and training a dedicated cost model for each group, allowing faster convergence and program selection. They show that prediction performance and learning efficiency can be improved depending on the learning strategy and cost model.

In optimizing process of *prior* and *posterior* subgraphs, the cost model of Bayesian code diffusion leverages strengths of the two approaches. First, during the optimization of *prior* subgraphs within each cluster, the cost model is trained using the execution data from *prior* programs across diverse clusters. We refer to this process as pre-training. Then, by tuning the *posterior* subgraphs, the cost model is further refined with execution data from *posterior* programs similar to their corresponding *prior* programs. We refer to this process as fine-tuning. This learning strategy enables the cost model to converge quickly while minimizing data bias.

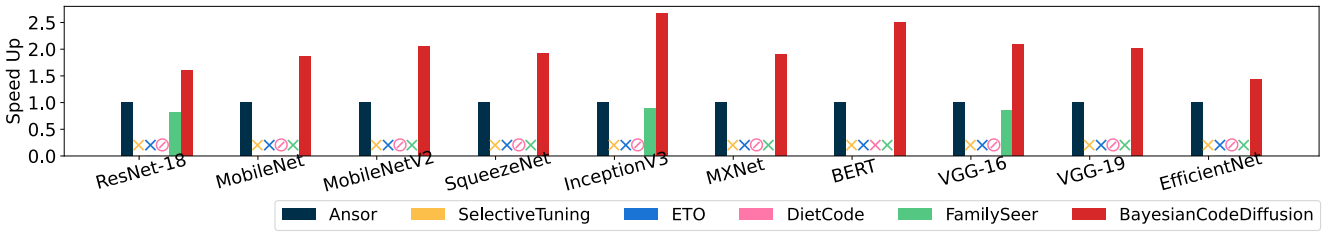
Importantly, we do not modify the cost model architecture or input features and use XGBoost [4] as the cost model, following Ansor [47]. Instead, we adjust the sequence in which execution (training) data is generated. Fig. 12 depicts subgraph optimization sequences of Ansor [47] and Bayesian code diffusion, generating data in different orders for cost model training. As shown in Fig. 5, for the clusters $\mathbb{C}_0, \mathbb{C}_1, \mathbb{C}_2$ and $\mathbb{C}_3$, *prior* subgraphs are selected as $\mathcal{G}_7, \mathcal{G}_3, \mathcal{G}_1$ and $\mathcal{G}_4$, respectively. Bayesian code diffusion optimizes each *prior* in proportion to the size of the cluster it belongs to, assigning greater importance to larger clusters. This contrasts with Ansor's cost model training sequence, which begins with a round-robin over randomly ordered subgraphs and then proceeds with a gradient-based selection strategy.

6 Experiments

We implement the proposed Bayesian code diffusion on Ansor [47] (`tvm@a340dbe`), which is available at GitHub



(a) Compilation (optimization) speedup on CPU (Intel Core i9-11900K)



(b) Compilation (optimization) speedup on GPU (Nvidia A6000)

Figure 13: The compilation (optimization) time speedup in generating deep learning programs with execution latencies equivalent to the best latencies achieved by Ansor [47], normalized to a baseline of 1.0 (the blue bars). The symbol '×' indicates that the corresponding method fails to generate a program with latency equivalent to Ansor's best, while '⊘' denotes that the corresponding method is not applicable to the target deep learning model or hardware (CPU/GPU).

repository[1], providing the detailed procedures. We evaluate it on an Intel Core i9-11900K@3.50GHz CPU and Nvidia A6000 GPU with a wide range of deep learning models, i.e., ResNet-18 [16], VGG-{16,19} [38], BERT [11], MobileNet [17], MobileNet-V2 [36], SqueezeNet-V1.1 [18], Inception-V3 [39], MXNet [5], and EfficientNet [40].

6.1 End-to-end Model Compilation

Fig. 13 presents the end-to-end optimization (compilation) speedup achieved by Bayesian code diffusion in generating deep learning programs with execution latencies equivalent to the best-latency programs produced by Ansor [47] on both CPU and GPU. The results are compared with alternative approaches, i.e., Ansor [47], FamilySeer [26], DietCode [46], ETO [12], and SelectiveTuing [3]. On the CPU, the proposed Bayesian code diffusion outperforms Ansor [47] by an average of 2.52× speedup (arithmetic mean) and other methods by an average of 1.95× speedup, with a maximum speedup of up to 3.31×. Similarly, on the GPU, Bayesian code diffusion outperforms all other methods by an average of 2.00×, with a maximum speedup of up to 2.79×. The result also shows that previous methods are either (1) unable to generate programs with the equivalent execution latencies, such as ETO [12] and FamilySeer [26], which are marked as × in Fig. 13 or (2) inapplicable to specific subgraphs (operators) or hardware platforms (CPU or GPU), such as DietCode [46], ETO [12], and SelectiveTuing [3] marked as ⊘. In particular, DietCode [46] is designed with GPU architectural features, such as thread blocks, and thus is limited to supporting only GPUs and a subset of operators, rather than the full range of operators and CPUs. Notably, DietCode does not implement the support for convolutional operations, and consequently, it is only evaluated on BERT. Similarly, ETO [12] is also tailored for GPU architectures, and SelectiveTuning [3] has demonstrated effectiveness only on GPUs, with no reported evaluation conducted on CPUs. Unlike these existing methods, which lack applicability, compatibility, optimality, and hardware independence, the proposed Bayesian code diffusion consistently generates programs with the minimum optimization times across various models and hardware (both CPU and GPU), enabling reliable, versatile, and efficient deep learning program optimization.

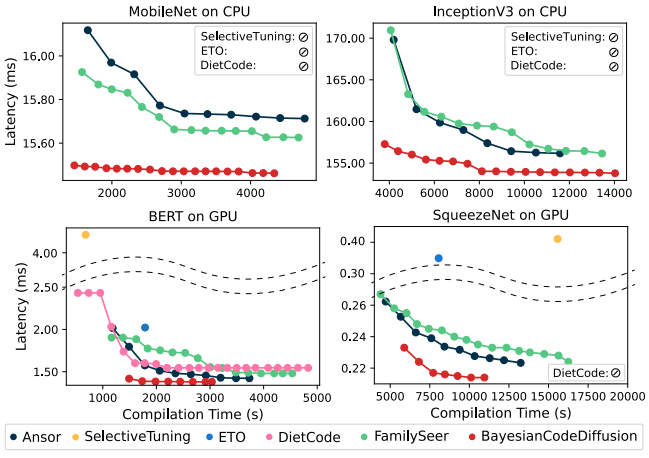


Figure 14: The execution latencies (ms) of the generated programs over compilation (optimization) time (s). The methods that are not applicable either to the target deep learning models or hardware (CPU/GPU) are indicated by the symbol '⊘'.

Fig. 14 presents program execution latencies over compila-

[1] https://github.com/eai-lab/BayesianCodeDiffusion

Table 4: The execution latency speedups of the programs generated by Bayesian code diffusion: the first diffused latency (left of the slash), measured at the first iteration, and the last diffused latency (right of the slash), measured after all iterations are completed. These are compared against alternative approaches, with Ansor's optimal latency normalized to 1.0. The symbol '⊘' denotes that the method is not applicable to the target deep learning model or hardware (CPU/GPU).

Hardware	Method	Model									
		ResNet-18	MobileNet	MobileNetV2	SqeezeNet	InceptionV3	MXNet	BERT	VGG-16	VGG-19	EfficientNet
CPU	SelectiveTuning	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘
	ETO	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘
	DietCode	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘
	FamilySeer	1.02 / 1.01	1.01 / 1.01	1.02 / 1.01	1.07 / 1.01	1.00 / 1.01	1.02 / 1.00	1.02 / 1.01	0.99 / 1.00	1.00 / **1.01**	1.00 / 1.00
	Bayesian code diffusion	**1.04 / 1.01**	**1.04 / 1.02**	**1.08 / 1.02**	**1.21 / 1.03**	**1.16 / 1.01**	**1.05 / 1.01**	**1.02 / 1.01**	**1.04 / 1.01**	**1.00 / 1.00**	**1.03 / 1.01**
GPU	SelectiveTuning	0.38 / 0.28	0.60 / 0.44	0.58 / 0.40	0.68 / 0.56	0.83 / 0.56	1.03 / 0.58	0.47 / 0.32	0.71 / 0.52	0.42 / 0.28	0.94 / 0.72
	ETO	0.94 / 0.68	0.55 / 0.41	0.63 / 0.43	0.80 / 0.65	1.18 / 0.80	0.78 / 0.44	1.04 / 0.70	0.87 / 0.63	0.69 / 0.47	1.04 / 0.80
	DietCode	⊘	⊘	⊘	⊘	⊘	⊘	0.86	⊘	⊘	⊘
	FamilySeer	1.00 / 1.02	0.99 / 0.98	1.03 / 0.98	1.00 / 1.00	1.00 / 1.01	1.02 / 0.98	1.10 / 0.97	1.00 / 1.01	1.00 / 0.93	1.02 / 0.99
	Bayesian code diffusion	**1.26 / 1.06**	**1.29 / 1.07**	**1.36 / 1.05**	**1.15 / 1.05**	**1.39 / 1.08**	**1.65 / 1.07**	**1.47 / 1.03**	**1.16 / 1.07**	**1.37 / 1.13**	**1.20 / 1.01**

tion (optimization) time on both CPU and GPU. It shows that Bayesian code diffusion provides enhanced performance in both program latency and compilation time when compared to alternative methods. For example, it provides optimization (compilation) speedups of 3.25× for MobileNet on CPU, 2.28× for InceptionV3 on CPU, 2.50× for BERT on GPU, and 1.91× for SqueezeNet on GPU, while generating programs whose execution latencies are consistently lower than those of other methods throughout the entire optimization. This is because Bayesian code diffusion and existing approaches, such as Ansor [47], explore the search space with distinct strategies. Notably, Bayesian code diffusion explicitly targets narrowed and more probable subsets of the search space. Although other approaches could find programs with comparable execution latencies to Bayesian code diffusion by continuously exploring the large entire search space, they would come at the cost of significantly increased time. As a result, Bayesian code diffusion can efficiently constrain and explore the search space, and identifies lower-latency programs more effectively compared to alternative methods, under the same time budget.

Tab. 4 summarizes the program latency speedup achieved by Bayesian code diffusion at the first iteration (i.e., first-diffused latency) and the last iteration of code diffusion (i.e., last diffused latency), respectively, in comparison with alternative methods. Following Ansor [47], Bayesian code diffusion allocates the total search time budget by dividing it among the subgraphs. After each subgraph is optimized through the pre-fine tuning process described in Sec. 5, the remaining time budget is used to further optimize the subgraphs in a gradient-based order [47]. We define the first-diffused latency as the program latency measured when all subgraphs have been tuned at least once. The last-diffused latency refers to the program latency obtained after fully utilizing the entire time budget to optimize all subgraphs. The first-diffused latency is compared against the program latency obtained when all subgraphs are tuned by each method. In contrast, the last-diffused latency is compared against the program latency achieved after each method has exhausted its entire search time budget. At the outset (Tab. 4, left of the slash), it begins from first diffused programs exhibiting the lowest latency across all cases, for instance 1.65× latency speedup for MXNet on GPU, compared to other approaches that either fail to generate programs with equivalent latencies or require significantly more time to achieve them. After full iterations of Bayesian code diffusion (Tab. 4, right of the slash), Bayesian code diffusion produces last diffused programs with the lowest program latency among all methods, e.g., 1.13× speedup for VGG-19 on GPU, while significantly reducing optimization time at the same time, as presented in Fig. 13. These results suggest that a series of mechanisms of Bayesian code diffusion, i.e., *prior* optimization, and code diffusion for *posterior* program generation, brings consistent improvements in both program compilation (optimization) time and the execution latency of the generated program.

6.2 Subgraph Cluster Optimization

Tab. 5 lists a set of operator sequences of several subgraph clusters in some deep learning models, and Fig. 15 presents their optimization speedup using Bayesian code diffusion, compared with Ansor [47]. As illustrated in the figure, the proposed Bayesian code diffusion optimizes deep learning operators to achieve latencies equivalent to Ansor's best results, with an average optimization speedup of 2.11×. Fig. 16 shows the optimization efficiency of Bayesian code diffusion on several subgraph clusters (i.e., clusters A, B, C, and D) over the optimization time, indicating that it generates programs with lower execution latency using reduced subgraph compilation (optimization) time compared to that of Ansor.

Table 5: A set of operation sequences in subgraphs.

Idx	Operator Sequence	Model	HW
A	conv2d_add_nn_relu	MobileNet	CPU
B	conv2d_NCHWc_add_add_nn_relu	MXNet	CPU
C	conv2d_NCHWc_add_nn_relu	InceptionV3	CPU
D	conv2d_add_nn_relu	SqueezeNet	GPU

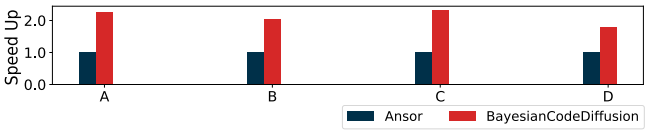


Figure 15: The optimization speedups on subgraph clusters in Tab. 5, compared against Ansor [47].

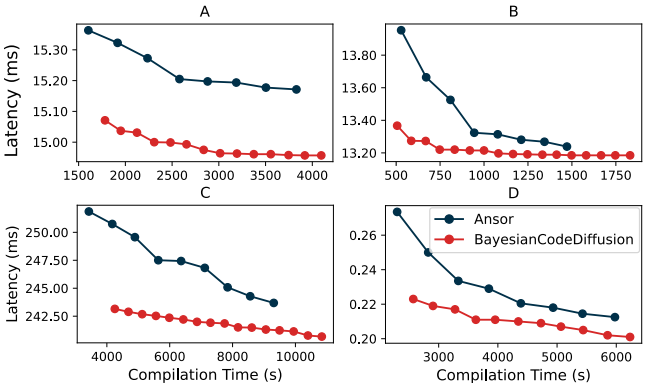


Figure 16: The latencies of the generated programs (ms) over the subgraph cluster compilation (optimization) time (s).

6.3 Cost Model

Fig. 17 plots the program execution latency over the compilation (optimization) time, comparing the proposed pre-training strategy and fine-tuning with Ansor [47] that does not apply it. The sequences of subgraph optimization for both cases are illustrated in Fig. 12, which generates training data samples in different orders for training the cost model. To evaluate the cost model as an independent variable, we optimize the programs using Ansor without using code diffusion in both experiments. The cost model is then trained on data samples produced in different orders. As shown in the figure, the proposed pre-training and fine-tuning method reduces compilation time by producing programs with lower latencies, without requiring any modifications to the cost model. This result suggests that the effectiveness of the cost model can be enhanced by first improving its generalization ability with various datasets and subsequently learning and making predictions on specific subgraph clusters. We expect further improvements could be achievable by incorporating advancements in enhancing the learning capacity and prediction accuracy of the cost model [2, 35, 44] into the proposed Bayesian code diffusion. We leave this exploration for future work.

6.4 Subgraph Sparsity

The performance of Bayesian code diffusion can be influenced by the degree of similarity among subgraphs of models.

In particular, if each subgraph generates a distinct sketch, it leads to an excessively large number of small clusters. As a result, the selected *prior* subgraph receives fewer tuning opportunities, and fewer *posterior* subgraphs can benefit from *prior* propagation, possibly reducing its effectiveness. Additionally, high operator diversity can negatively affect the cost

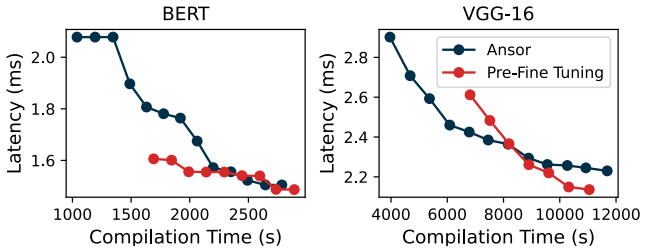


Figure 17: The latency (ms) of generated programs for BERT [11] and VGG-16 [38] over the compilation (optimization) time (s), under different training orders of the cost model as illustrated in Fig. 12.

Table 6: A comparison between sketch sparsity ($\mathbb{S}_S$), operator sparsity ($\mathbb{S}_O$), and program optimization speedup.

Network	$\mathbb{S}_O$	CPU		GPU	
		$\mathbb{S}_S$	SpeedUp	$\mathbb{S}_S$	SpeedUp
ResNet-18	0.83	0.83	2.63	0.83	1.60
MobileNet	0.14	0.57	3.25	0.14	1.85
MobileNetV2	0.44	0.76	2.62	0.71	2.05
SqueezeNet	0.22	0.43	3.30	0.22	1.92
InceptionV3	0.22	0.53	2.42	0.22	2.67
MXNet	0.62	0.86	2.51	0.62	1.91
BERT	0.38	0.50	2.75	0.50	2.50
VGG-16	0.40	0.93	2.48	0.87	2.08
VGG-19	0.40	0.93	1.33	0.87	2.01
EfficientNet	0.66	0.88	1.85	0.88	1.43

model's pre-training and fine-tuning strategies. Even when sketches differ, subgraphs that share the same operator tend to produce similar program data. Consequently, fine-tuning results from one cluster can positively influence others that, despite having different sketches, share the same operator. However, when operator diversity is high, such cross-cluster generalization becomes limited, reducing the effectiveness of the cost model's learning strategy.

To quantify the diversity of sketch generation and operator across subgraphs, we define sketch sparsity and operator sparsity as shown in Eq. (10), where G_{all} is the total number of subgraphs, $\max(|G_S^i|)$ is the number of subgraphs that belong to the largest cluster obtained via sketch-based clustering, and $\max(|G_O^i|)$ is the number of subgraphs containing the most frequent operator in the model. A higher sparsity value indicates lower subgraph similarity. Tab. 6 compares sketch sparsity, operator sparsity, and optimization speedup of Bayesian code diffusion in Fig. 13 across different models.

$$\mathbb{S}_S = 1 - \max(|G_S^i|)/G_{\text{all}},\ \mathbb{S}_O = 1 - \max(|G_O^i|)/G_{\text{all}} \quad (10)$$

Fig. 18 presents the Pearson correlation [13] between subgraph sparsity (i.e., sketch sparsity and operator sparsity) and average program optimization speedup on various models on CPU and GPU. Overall, the sketch sparsity shows a stronger correlation with optimization speedup on CPU, while operator sparsity is more strongly correlated on GPU. This indicates that *prior* propagation has a large impact on compilation time on CPU, whereas pre-training and fine-tuning strategy in cost model are more impactful on GPU.

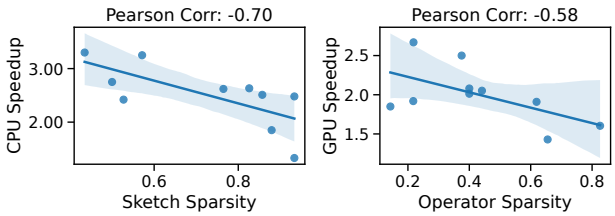


Figure 18: The Pearson correlation [13] between subgraph sparsity and speedup on CPU and GPU. On CPUs, speedup is correlated more strongly with sketch sparsity, whereas on GPUs, it is more closely associated with operator sparsity.

6.5 Prior Selection

We evaluate the *prior* selection described in Sec. 4.2. Specifically, we compare the program execution latency of the first diffused code in Bayesian code diffusion; the *prior* subgraph is selected based on either high or low similarity to other subgraphs in the same cluster. Fig. 19 presents the mean latency of l_h and l_l across various models in Fig. 13, where l_h and l_l represent the program execution latency of the first diffused code obtained by selecting *prior* subgraphs with high and low similarity, respectively. In general, selecting a *prior* subgraph with high similarity enables Bayesian code diffusion to find lower-latency programs across both CPU and GPU. On average, the normalized latency l_h/l_l is 0.99 on CPU and 0.91 on GPU, indicating consistent performance improvement of high-similarity *prior* subgraph over low-similarity *prior* subgraph. Therefore, Bayesian code diffusion tends to be more effective when selecting the subgraph with the highest similarity to others as the *prior* subgraph within the cluster.

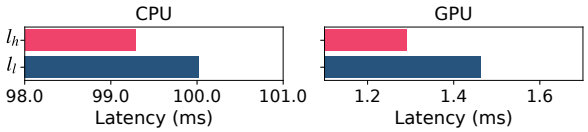


Figure 19: A comparison of l_h and l_l on CPU and GPU, where l_h and l_l denote the program execution latency of the first diffused code optimized using high- and low-similarity *prior* subgraphs, respectively. The bar lengths indicate the mean value of execution latencies of various deep learning models.

7 Discussions and Limitations

From Hypothetical Concept to Implementation. Applying the proposed concept of Bayesian code diffusion to the deep learning program optimization problem requires several considerations and presents some complexities in implementation. For instance, the true prior distribution $f(\theta_p) = \mathcal{N}(\theta_p^*, \sigma_p^2)$ is unknown and must be effectively approximated to conduct precise code diffusion for the *posterior* optimization. Although the optimal *prior* parameter θ_p^* is empirically estimated in our implementation by allocating a large search time budget proportional to the cluster size for sufficient search, there is no guarantee that the identified θ_p^* is truly optimal. While Eq. (4) in Sec. 3.1 provides a confidence measurement on the *posterior* optimal parameter θ_s^*, a similar quantity needs to be devised for the optimal *prior* parameter θ_p^* to improve the overall performance of subsequent code diffusion. Furthermore, implementing code diffusion for the *posterior* program optimization may not be straightforward in practical implementations. In this work, we demonstrate a concrete implementation of code diffusion for Ansor [47]; however, different deep learning compilers or optimization mechanisms may require modifications of the code diffusion approach, distinct from the approach presented here. We will investigate to bridge the gap between the hypothetical concept of Bayesian code diffusion and its practical implementations, aiming to develop a generalized solution that can address or mitigate these implementation challenges.

Prior Selection. Selecting the optimal *prior* subgraph within a subgraph cluster during online program optimization is a challenging task. That is because the *prior* subgraph capable of generating the optimal programs can only be identified after the full compilation (optimization) of the target deep learning model. In this work, we estimate the most suitable *prior* during the optimization process by measuring the tensor shape distances between subgraphs in a cluster in an online manner. Although this approach results in substantial improvements in both optimization time and program execution latency, we have observed that superior *prior* subgraphs exist in certain cases, which could further enhance program optimization performance. We will study alternative approaches that can more accurately select the *prior* subgraph to facilitate enhanced code diffusion for the *posterior* parameter optimization.

Cost Model. While Bayesian code diffusion proposes and employs pre-training and fine-tuning of the cost model, which enhances the program optimization performance, additional improvements could be achieved by enhancing the cost model itself. For instance, in this work, the cost model or learning algorithm is utilized as presented in Ansor [47] without modification. However, more advanced cost models, such as neural network-based approaches [44], could be employed to enhance the model prediction accuracy. Regarding learning strategies, active learning [2], which selects and focuses on learning from data examples expected to contribute to model performance, can be used to enhance both prediction accuracy and training efficiency. Additionally, we can leverage a large pre-collected dataset as pre-training data in an offline manner [35], followed by fine-tuning the cost model in an online context. Therefore, it is worth investigating how improvements to the cost model, including modeling methods and learning strategies, affect the performance of the proposed Bayesian code diffusion in a synergistic manner. However, we generally conjecture that improvements to the cost model could help further enhance program optimization, as these mechanisms are orthogonal to one another.

Applicability to Other Deep Learning Compilers. The core idea behind Bayesian code diffusion is that structurally

similar subgraphs can share optimization patterns, enabling more efficient program tuning. This insight can extend beyond Ansor [47], to other deep learning program optimization frameworks such as MetaSchedule [37] and TASO [20], expanding its applicability. MetaSchedule [37], a stochastic search framework, probabilistically samples transformation rule parameters. Bayesian code diffusion can be integrated into MetaSchedule, where *posterior* program distributions follow *prior* distributions, or their parameters are diffused from *prior* programs. For TASO [20], a superoptimizer based on graph substitutions performs graph transformations independently by generating and comparing candidates. While TASO does not exploit subgraph similarity, its search space can be reduced by identifying and matching common subpatterns.

8 Related Work

Subgraph Similarity. Similar to Bayesian code diffusion, some existing works group subgraphs. TransferTuning [14] transfers an optimized schedule from one model to another running on CPU based on model similarity. However, it requires a pre-optimized schedule of a separate model, which differs from the proposed approach not requiring it. Instead, the *prior* is propagated in an online manner within the same model, which is applicable to both CPU and GPU. FamilySeer [26] groups subgraphs with identical operator sequences and assigns a cost model to each. However, the narrow dataset generated by a single group often leads to suboptimal program performance. The proposed approach addresses this by pre-training and fine-tuning the cost model, accelerating program optimization while ensuring optimal performance. The proposed approach can also be applied to reduce search spaces in dynamic tensor program optimizations, e.g., Nimble [21] and DietCode [46], which construct a shape-generic search space. SelectiveTuning [3] also clusters subgraphs and transfers schedules from representative subgraph to others. In contrast to this method, which provides suboptimal performance as it modifies the transferred schedule only once using pre-defined rules, the proposed approach employs iterative and adaptive code diffusion, producing optimal programs.

Efficient Search Space Exploration. Many studies try to facilitate efficient exploration of program search spaces. Being orthogonal to the proposed Bayesian code diffusion, they can be applied to further accelerate optimization in the reduced search space derived from the *prior* programs. For example, Chameleon [1] explores search spaces using reinforcement learning, and ALT [42] minimizes hardware measurements while exploring optimal compile time with active learning. Dynamic gradient descent search [24] improves the performance of the program by replacing traditional random sampling. Some works aim to address the inefficiencies of auto-schedulers. DynaTune [45] efficiently allocates time resources to optimization tasks, and AdaTune [25] reduces the hardware measurements needed to stabilize the performance of the program. However, these approaches do not utilize similarity between subgraphs, missing opportunities for more efficient exploration and program reuse. Thus, Bayesian code diffusion can complement them. Thus, integrating Bayesian code diffusion can reduce redundant searches by reducing exploration spaces, improving exploration efficiency, reducing hardware measurements, and enhancing cost model accuracy, leading to more effective deep learning program optimization.

Cost Model. Cost models are typically trained online during compilation using data generated via hardware measurements [6, 47]. BALTO [2] reduces training data through diversity-based active learning, and TLP [44] uses a neural network and extracts features from primitives to ensure portability across various hardware and efficiently explore programs with limited data. In contrast, offline learning utilizes a pre-trained cost model using pre-collected data to reduce compile time. This includes approaches such as Tenset [48] and One-shot tuner [35] that train a neural network cost predictor offline. Although offline learning reduces compile time, it requires a substantial amount of data to train the model, which may not be available for heterogeneous hardware. This contrasts with the proposed approach for online training. In this work, we introduce the concept of online pre-training of the cost model using the *prior* s of different clusters, which is fine-tuned on cluster-specific data, one at a time. It enables the model to ensure generalizability while improving the predictions for each cluster, without modifying the cost model.

Large Language Models. Advances in large language models (LLMs) have prompted their adoption in code generation tasks [8, 9, 43]. In particular, TLM [43] proposes a tensor language model for deep learning program optimization and departs from traditional program sampling methods by reconstructing the search space through a generative compiler based on LLMs. However, this approach requires extensive computational resources due to the pre-training of huge models like GPT-2 [31] with large-scale datasets. Additionally, this offline learning paradigm struggles with heterogeneous hardware, differing fundamentally from the online learning approach adopted in Bayesian code diffusion.

9 Conclusion

We introduce Bayesian code diffusion, which enhances the auto-tuning efficiency for deep learning (tensor) programs by reformulating optimization as a distribution-based problem using Bayesian principles. By diffusing well-optimized program code as a *prior* across similar subgraphs, it enables an efficient search for optimal parameters for *posterior* programs compared to random search methods. Additionally, pre-training and fine-tuning the cost model improves prediction accuracy. Implemented on Ansor, the proposed Bayesian code diffusion accelerates compilation times and reduces the program latency for deep learning models on both CPUs and GPUs, outperforming state-of-the-art auto-tuning methods.

Acknowledgments

This work was supported by the Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00508465) and Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2020-II201336, Artificial Intelligence Graduate School Program(UNIST)).

References

[1] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743*, 2020.

[2] Jun Bi, Xiaqing Li, Qi Guo, Rui Zhang, Yuanbo Wen, Xing Hu, Zidong Du, Xinkai Song, Yifan Hao, and Yunji Chen. Balto: fast tensor program optimization with diversity-based active learning. In *The Eleventh International Conference on Learning Representations*, 2022.

[3] CH Yu. SelectiveTuning. https://github.com/apache/tvm/issues/4188, 2019.

[4] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[8] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

[9] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[10] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[12] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. Eto: accelerating optimization of dnn operators by high-performance tensor program reuse. *Proceedings of the VLDB Endowment*, 15(2):183–195, 2021.

[13] David Freedman, Robert Pisani, and Roger Purves. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.

[14] Perry Gibson and José Cano. Transfer-tuning: Reusing auto-schedules for efficient tensor program code generation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 28–39, 2022.

[15] Google. Xla - high level optimizer. GitHub repository, 2024.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[18] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[19] Intel Corporation. *Intel® Math Kernel Library (Intel® MKL)*, 2024. Version 2024.0.

[20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[21] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems*, 33:8343–8354, 2020.

[22] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[23] Chris Leary and Todd Wang. Xla: Tensorflow, compiled, 2017.

[24] Chendi Li, Yufan Xu, Sina Mahdipour Saravani, and Ponnuswamy Sadayappan. Accelerated auto-tuning of gpu kernels for tensor computations. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 549–561, 2024.

[25] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: Adaptive tensor program compilation made efficient. *Advances in Neural Information Processing Systems*, 33:14807–14819, 2020.

[26] Mingzhen Li, Hailong Yang, Shanjun Zhang, Fengwei Yu, Ruihao Gong, Yi Liu, Zhongzhi Luan, and Depei Qian. Exploiting subgraph similarities for efficient auto-tuning of tensor programs. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 786–796, 2023.

[27] Jonas Mockus and Jonas Mockus. *The Bayesian approach to local optimization*. Springer, 1989.

[28] Pranati Modumudi, Xiyou Zhou, and Sunghyun Park. Unveiling source of performance variance on search-based compiler optimization. In *Machine Learning for Computer Architecture and Systems 2022*.

[29] NVIDIA Corporation. *cuBLAS: The NVIDIA CUDA Basic Linear Algebra Subroutines library*, 2024.

[30] NVIDIA Corporation. NVIDIA cuDNN. https://developer.nvidia.com/cudnn, 2024. Version X.Y.Z.

[31] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[32] Sivaramakrishnan Rajaraman, Ghada Zamzmi, Feng Yang, Zhaohui Liang, Zhiyun Xue, and Sameer Antani. Semantically redundant training data removal and deep model classification performance: A study with chest x-rays. *arXiv preprint arXiv:2309.09773*, 2023.

[33] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level compiler for deep learning. *arXiv preprint arXiv:1904.08368*, 2019.

[34] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[35] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. One-shot tuner for deep learning compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 89–103, 2022.

[36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[37] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[38] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[39] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[40] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[41] Pradnya A Vikhar. Evolutionary algorithms: A critical review and its future prospects. In *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*, pages 261–265. IEEE, 2016.

[42] Xi Zeng, Tian Zhi, Zidong Du, Qi Guo, Ninghui Sun, and Yunji Chen. Alt: Optimizing tensor compilation in deep learning compilers with active learning. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 623–630, 2020.

[43] Yi Zhai, Sijia Yang, Keyu Pan, Renwei Zhang, Shuo Liu, Chao Liu, Zichun Ye, Jianmin Ji, Jie Zhao, Yu Zhang, and Yanyong Zhang. Enabling tensor language model to assist in generating High-Performance tensor programs for deep learning. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 289–305, Santa Clara, CA, July 2024. USENIX Association.

[44] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 833–845, 2023.

[45] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. Dynatune: Dynamic tensor program optimization in deep neural network compilation. In *International Conference on Learning Representations*, 2021.

[46] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems*, 4:848–863, 2022.

[47] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[48] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.



Training with Confidence: Catching Silent Errors in Deep Learning Training with Automated Proactive Checks

Yuxuan Jiang Ziming Zhou Boyu Xu Beijie Liu Runhui Xu Peng Huang

University of Michigan

Abstract

Training deep learning (DL) models is a complex process, making it prone to silent errors that are challenging to detect and diagnose. This paper presents TRAINCHECK, a framework that takes a proactive checking approach to address silent training errors. TRAINCHECK automatically infers invariants tailored for DL training. It uses these invariants to proactively detect silent errors during the training process while providing debugging help. To evaluate TRAINCHECK, we reproduce 20 *real-world* silent training errors with diverse root causes. TRAINCHECK successfully detects 18 errors within a single training iteration. It also uncovers 6 unknown bugs in popular training libraries that lead to silent errors.

1 Introduction

Training deep learning (DL) models has become integral for many application domains [8, 17, 24]. DL training, however, is a complex process involving multiple steps and layers of components such as user code, compiler, training framework, optimization libraries, drivers, and distributed systems. Moreover, these components undergo frequent updates [9, 13] due to the rapid pace of DL research. Consequently, training jobs are prone to errors from various sources [51].

To make matters worse, while some errors cause immediate job failures and are relatively easy to identify (*e.g.*, GPU out of memory or illegal argument exceptions), many others are *silent or latent*. Such errors do not cause obvious training disruptions but eventually produce suboptimal/incorrect models or cause noticeable failures much later.

Figure 1 shows a *real-world* silent training error in HuggingFace's training of BLOOM-176B [4]. DeepSpeed's `BF16Optimizer` was used in this training task, which had a logic bug in gradient clipping. This bug did not trigger any exception but caused parts of the model to diverge silently across GPUs, a problem that went undetected for 10 days.

Detecting such silent errors is inherently difficult due to the lack of clear signals. When they are detected, substantial training resources have been wasted. The problem is further compounded by the trend of training large models using massive resources. Moreover, during diagnosis, developers are often left in the dark and have to take a trial-and-error approach [53], such as tweaking hyperparameters and rerunning the task while the real root causes are elsewhere, which is tedious and

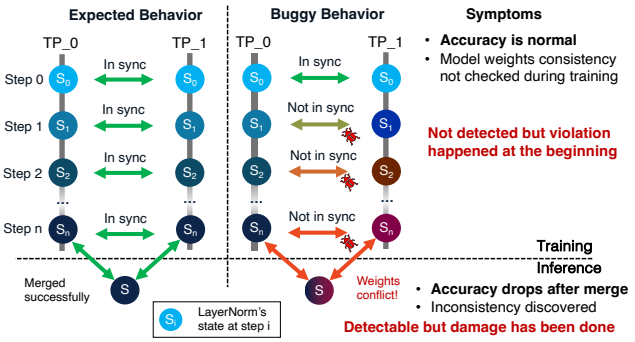


Figure 1: Silent error in BLOOM-176B training.

time-consuming. Indeed, developers have expressed frustration with diagnosing these silent errors [7, 20, 37, 38, 46, 47].

Current practices rely on high-level model evaluation signals such as loss, accuracy, and gradient norms [1, 5]. While these signals provide useful information about overall training progress, they are not designed for error detection. They are often noisy and evaluated only periodically, leading to missed or delayed detection. Even when anomalies are observed, they provide little diagnostic value for identifying the root cause. The aforementioned BLOOM-176B training error, for example, did not manifest as abnormal loss or accuracy, and it took developers 12 days to diagnose and resolve the problem [4, 6]. A few static solutions exist to detect certain errors with specific root causes, such as tensor shape mismatches [22], but they fail to cover many other types of silent training errors.

Our insight is that a fundamental reason behind the challenges is the lack of *proactive checks* to continuously validate a training task. We call such checks, *training invariants*, which are rules that should hold throughout the training. Violation of a training invariant indicates potential errors. The concept of invariants is not new and has been widely studied in traditional software. However, existing tools such as Daikon [16] focus on low-level variable relationships (e.g., `var1 > var2`) and fail to capture the high-level semantics of silent errors in DL training, making them ill-suited for this domain.

Based on this insight, we design an end-to-end framework, TRAINCHECK, that enhances a DL training job with training invariants to proactively detect a wide variety of silent errors. Compared to generic high-level signals, the training invariants TRAINCHECK introduces can more accurately and quickly detect anomalies. Upon detection, TRAINCHECK provides debugging information indicated through the violated

invariant and relevant traces. Compared to static approaches, TRAINCHECK can provide stronger assurance. Its checks capture misbehavior caused by various root causes. It operates continuously with a running training task, enabling the detection of real errors in deployment, which may only occur under specific datasets, environments, or scales.

In designing TRAINCHECK, we aim to explore two key research questions. First, *what kind of training invariants are effective* to address silent training errors? We initially hypothesized that, due to the stochastic nature of DL training, the invariants would need to encode probabilistic predicates, which can be complex and unstable. Our further investigation leads to the insight that, by properly choosing the level of behavior to observe, a training invariant can be made simpler and more precise. Non-determinism is an artifact of checking at too high of a level. The invariants should operate at a level below model evaluation signals, but not as low as traditional software invariants. In the BLOOM-176B example, an effective training invariant is roughly: the weights of certain layers should stay consistent across tensor parallelism (TP) ranks. We thus focus on such rule-level training invariants.

Second, *how can we automatically obtain and check training invariants*? The above invariant description is informal. The actual invariant must be concrete and detailed enough to be directly *checkable* for a specific training task. It is impractical to ask developers to manually write and maintain such invariants. Deep learning frameworks and training practices evolve rapidly, introducing new APIs, model architectures, and algorithms that make handwritten invariants difficult to catch up. Moreover, invariants must match precisely with the actual implementation, not just high-level intent.

Automated inference is needed, but this is challenging. For example, to infer the above invariant, we need to collect information about each worker's role (*e.g.*, TP rank), tensor properties (replicated or partitioned), and observed values. The conditions under which this invariant applies are subtle: it only holds in distributed training when using tensor parallelism and LayerNorm, and only for weights *replicated* across TP ranks. This makes identifying the correct *precondition* critical—for example, the tensor must have `tensor_model_parallel=False`. These kinds of invariants go beyond what traditional invariant inference tools like Daikon [16] are designed to handle.

TRAINCHECK is designed to address these unique challenges and automatically infer concrete, checkable training invariants for assorted DL training tasks written on top of popular frameworks such as PyTorch [2] and DeepSpeed [40]. TRAINCHECK first instruments a given DL training program to collect traces. To achieve high usability and low overhead, we take a monkey-patching approach to dynamically inject the instrumentation code for framework API invocations and a proxy-based approach to intercept state updates.

To infer invariants from the collected traces, TRAINCHECK defines a set of generic relation templates. It uses an efficient algorithm that generates hypotheses based on a relation template and validates the hypotheses in the traces to generate invariants. TRAINCHECK further designs an algorithm to deduce the precondition, if any, for each invariant.

To use the inferred invariants for detecting silent training errors in a specific training pipeline, TRAINCHECK selectively instruments the pipeline for only information relevant to the invariants. A verifier continuously validates the traces from the instrumented training task to check violations.

Unlike traditional invariant inference tools that operate within a single program, a unique feature of TRAINCHECK is its ability to generate invariants *transferable* across different training programs and even different libraries. This makes TRAINCHECK broadly applicable and adaptable. Consequently, we can leverage high-quality DL training pipelines, such as those found in tutorials and example repositories, *e.g.*, PyTorch examples [35], to infer the invariants and apply them to other training programs. This helps with both improving invariant accuracy and aggregating effective invariants.

For evaluation, we collect and reproduce 20 *real-world* silent training errors with diverse root causes. TRAINCHECK detects 18 cases within a single training iteration while providing debugging hints. TRAINCHECK additionally uncovers 6 previously unknown errors, all of which have been confirmed with 3 being fixed. We also share our experiences applying TRAINCHECK to various DL training scenarios and highlight challenges and opportunities in this domain. TRAINCHECK is open sourced at https://github.com/OrderLab/TrainCheck.

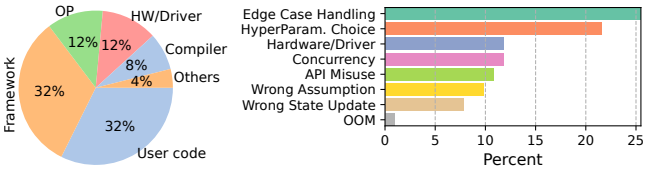
The contributions of this paper are as follows:

- We investigate the notorious yet under-explored problem of silent errors in DL training and conduct an empirical study to shed light on their characteristics.
- We propose an approach that uses training invariants to proactively validate DL training and catch silent errors.
- We design and implement TRAINCHECK, which, to the best of our knowledge, is the first framework that automatically infers and checks invariants tailored for DL training tasks.
- We evaluate TRAINCHECK in detecting real-world silent training errors and exposing unknown errors.

2 Silent Errors in DL Training

To gain a deeper understanding of silent training errors, we conduct an empirical study on *real-world* errors that DL practitioners encountered. We aim to provide insights regarding their root causes, impact, and current detection methods.

Methodology To ensure the study is representative, we inspect diverse sources: (1) GitHub issues from popular libraries such as PyTorch [36] and DeepSpeed [12], (2) discussion forums such as StackOverflow [41] and the PyTorch forums [34], and (3) papers and blog posts from DL practitioners [6,52]. For GitHub, we search closed issues labeled or containing the keywords "correctness" or "silent". For discussion forums, we search for posts with "silent" or "bug" in the title or description. We also examine DL training papers and blog posts from large organizations such as HuggingFace [4],



(a) Location. (b) Type.

Figure 2: Root cause locations and types of the studied errors.

Bloomberg [45], and Meta [14, 52], particularly those discussing widely deployed LLMs like Llama3 [14], OPT [52], and BLOOM [4]. These sources often describe errors arising in industrial-scale training pipelines. Many of the candidate reports from GitHub and user forums are incomplete or poorly understood, as silent errors are often difficult to diagnose. We carefully inspect each case and curate a set of high-quality instances that are well-documented, diagnosed with clear root causes, likely reproducible, and impactful.

In total, we collect 88 silent errors with known root causes. Among the sources, 70 are GitHub repository issues, 16 from discussion forums such as StackOverflow and the official PyTorch Forums, and 2 from industrial reports [4, 52].

2.1 Analyses and Observations

Diverse Root Causes The studied errors are caused by defects in a wide range of components involved in DL training, including user code, framework, compiler, mathematical operators for different hardware or from optimization libraries, and underlying system components such as driver or hardware.

Figure 2a shows the distribution of the locations of the root causes. The majority of the errors are caused by user code (32%) and framework (32%), followed by mathematical operations (12%), hardware (12%), compiler (8%), and other factors (4%). User code may contain missing or incorrect API calls or poorly chosen hyperparameters. Frameworks may have bugs in their Python components, often involving high-level algorithm logic. Mathematical operations such as matrix multiplication can produce inaccurate results. Hardware failures are usually caused by driver or device faults that lead to communication errors or memory corruption. Compiler failures occur when Torch Dynamo, PyTorch's JIT compiler, is unable to compile the pipeline correctly.

Figure 2b shows the root cause types for the studied cases. For user-code-induced silent errors, we identify two primary categories: (1) *Incorrect Implementation* and (2) *Inappropriate Design Choices*. The first includes incorrect, missing, or inconsistent API calls or parameter updates. For example, one error is caused by a missing `zero_grad()` inside the training loop, resulting in noisy gradients; another stems from initializing the optimizer before model transformations, leaving the optimizer with incorrect parameters to update. The second category involves flawed design choices, such as selecting an unsuitable loss function, setting an overly aggressive dropout rate, or using a problematic data processing pipeline. While individual components may appear correct, their interaction often leads to numerical instability or divergence. For instance, BLOOM developers encountered repeated loss explosions when training with the float16 data type, which was mitigated by switching to bfloat16 and applying gradient clipping.

Iter	Type	Loss Diff	PPL Diff	Diff (Loss/PPL)
2000	Valid	+1.14%	+1.43%	+0.014 / +0.050
2000	Test	+2.74%	+3.30%	+0.032 / +0.107
4000	Valid	+3.05%	+3.36%	+0.033 / +0.099
4000	Test	+4.67%	+4.79%	+0.047 / +0.131

Table 1: Reproducing DeepSpeed-1801 (the root cause of BLOOM-176B silent training error) in a small transformer-based language model. Due to resource constraints, we trained for only 2000 and 4000 iterations. However, the impact of the error—observed as differences in loss and perplexity due to weight merging—is already noticeable and increases with the number of training iterations.

Severe Consequences The impact of silent errors can be enormous. DL training tasks are often run on many GPUs for a long time. This is the case for the motivating example in Figure 1, which is a large-scale training task that involves 384 A100 GPUs for 3.5 months. Discovering errors late in the training result in significant waste of expensive resources.

The majority of the studied errors result in either suboptimal or incorrect models. In the former case, users may choose to accept the degraded outcome to avoid paying the expensive retraining cost, but must live with lower model performance. In the latter case, rerunning the job leads to a significant waste of costly training resources. Many errors also introduce severe performance impacts. For example, in PyTorch-Forum-84911, the data processing code in the user program mistakenly resizes the input images to 1024×1024 instead of the expected 224×224, significantly increasing per-iteration training time. Moreover, due to their silent nature, the effects of these errors often persist and accumulate over time.

Challenging Detection and Diagnosis Unlike other DL training failures that trigger explicit exceptions (*e.g.*, Illegal Argument) or terminate jobs, silent training errors are challenging to detect. In many of our studied errors, although the root cause is triggered early on, the error remains undetected for a long time. Developers currently mainly rely on high-level model evaluation metrics such as loss or accuracy. However, silent errors often do not cause immediate anomalies in such evaluation metrics. For example, in PyTorch-Forum-84911, the loss and accuracy metrics are still changing over time. For large-scale training jobs, it can take hours or even days for a silent error to show obvious signs of anomalies in these metrics. Moreover, these metrics can be highly noisy in real-world training tasks. Using them for detection can easily create false alarms. When a silent error causes a performance impact, it can also be difficult for developers to tell whether or not the long training iteration is expected. Even after the error is detected, developers often have no clear clues of the root cause. The diagnosis process is onerous and often based on

ad-hoc trial-and-error changes (e.g., to hyperparameters) and rerunning the training job to check if the error is fixed, leading to both developer productivity loss and wasted resources.

2.2 Case Studies

We describe two representative cases from our study.

BLOOM-176B training (DeepSpeed-1801) This is the example described in § 1, which occurred in HuggingFace's training of BLOOM-176B in 2022 [4]. Due to the large model size, 3D parallelism [31] was used to partition the model across multiple GPUs and nodes. Conceptually, tensor parallelism partitions the model by splitting individual layers into multiple GPUs. Thus the same layer on different GPUs is not the same. However, in implementation, due to the communication cost, certain layers that do not impose a memory bottleneck are not partitioned. In the Megatron-style tensor parallelism [31], LayerNorm layers are not partitioned as their size is small (<1% of the model weights) compared to the other layers, like attention layers. Thus, optimizers must be aware of this partition scheme, carefully distinguish between different kinds of layers, and perform updates accordingly. This leads to complex logic in the optimizer.

In this specific training task, DeepSpeed's BF16Optimizer was used. It had a bug that causes the gradient clipping to be enabled on only the first GPU on layers that are not partitioned. This caused the LayerNorm layers' weights to silently diverge, as these layers were updated with different gradients.

This bug neither triggers exceptions nor immediately affects loss or accuracy. Only when the model partitions across GPU need to be merged into one checkpoint file will users realize that the model has diverged. This happens in two scenarios: (1) when training completes, and the model needs to be served or further fine-tuned; (2) during training, there is a need to change the parallelism configuration, e.g., due to failure of a GPU. The developers of BLOOM-176B were lucky enough to catch this error before the model was served in production or further fine-tuned while investigating another error. The detection took 10 days, and it required 9 additional days to merge the weights and mitigate the impact of divergence.

We conduct an experiment to further confirm the impact of this error on a small scale. Specifically, we train a small transformer-based language model using the CodeParrot clean dataset [10], with 4 tensor parallel (TP) ranks and 2 data parallel (DP) ranks. The results are shown in Table 1.

At the initial steps (2000 and 4000) iterations, the loss and the perplexity of the model on the validation and test set are both noticeably affected by merging the weights, and the difference increases with the number of training iterations.

PyTorch-115607 Another interesting case is PyTorch-115607. It showcases situations when a silent training error can be detected, but the diagnosis is frustrating. `torch.dynamo` [2] is a new feature in PyTorch that provides a Just-In-Time (JIT) compiler for unmodified PyTorch programs. It transparently swaps bytecode to be executed with a more optimized version and thus can speed up the training process. It uses guards to ensure that when certain conditions change (e.g., the shape of a tensor changes), the bytecode is recompiled to reflect the changes. Ideally, every critical variable that can affect training should be guarded. However, in this case, a guard is missing, which causes the model to not update if a developer decides to only do a forward pass without a backward pass in a certain iteration. This error can be detected since the entire model is not updating. However, diagnosing it can be problematic: if the training program does not dump per-iteration loss and accuracy, the developer would have no idea where the model stopped updating.

2.3 Implications

Our study underscores the significant challenges posed by silent errors in DL training. It also reveals implications for designing solutions to address such errors.

Manual methods for detecting and diagnosing silent errors are time-consuming, ineffective, and impractical, given the scale and complexity of the DL training system stack. Automated approaches are essential to reduce the burden on developers while improving the reliability of training pipelines.

Monitoring high-level signals such as loss and accuracy is insufficient. These metrics are often noisy and periodic, leading to missed or delayed detection. They are also prone to false alarms because it is difficult to distinguish between expected fluctuations and true anomalies. Moreover, they provide few clues for debugging. Fundamentally, they are not designed for error detection. A true detection solution is required and needs to operate at a deeper level than such signals.

The diverse locations and root causes of silent errors call for systematic approaches. While point solutions, such as using differential testing [27, 33] to find bugs in DL compilers and frameworks, have been proposed, they offer only limited help in addressing the broad categories of silent errors.

Eliminating all bugs statically is ideal, but it is nearly impossible due to the complexities involved in DL training. Many bugs tend to only trigger in a specific environment, dataset, or scale. Continuous monitoring of a DL training task provides a safety net to detect silent errors at runtime.

3 System Design

Motivated by our study, we design TRAINCHECK, a framework that takes a proactive validation approach to quickly and reliably detect silent DL training errors with a wide variety of root causes, while providing diagnosis clues.

The designs of TRAINCHECK are guided by two key insights we have from analyzing real-world cases. First, while the symptoms of silent training errors take time to become visible and often appear non-deterministic in generic, high-level signals such as loss, the root causes of many errors are triggered early on and detectable through specific, lower-level checks. Second, seemingly unrelated DL training programs

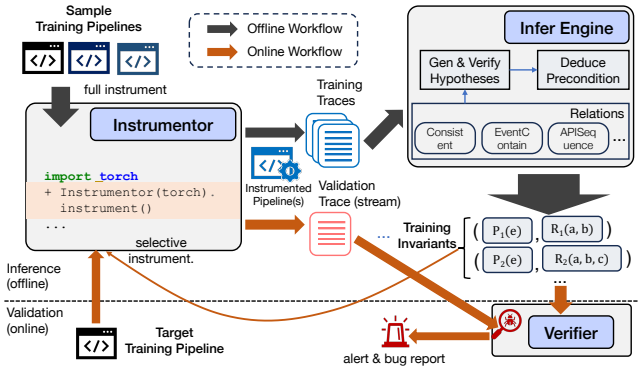


Figure 3: Workflow of TRAINCHECK.

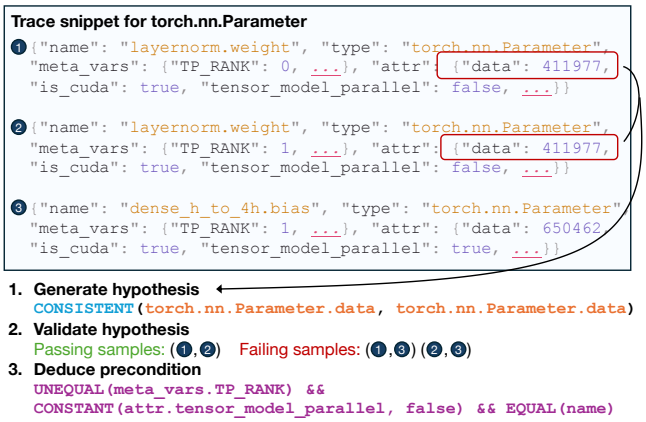


Figure 4: The invariant with preconditions TRAINCHECK infers for Bloom-176B error (§ 2.2), and the simplified trace it emits and uses.

can share similar correctness properties due to the heavy reliance on external libraries and similar training methods.

Based on these insights, TRAINCHECK introduces the notion of *training invariants*, which are rules that should hold during training, *e.g.*, the model weights should stay consistent across distributed workers. These rules capture the semantics and correctness properties specific to a training task. Interestingly, we observe that a training invariant often only applies under specific conditions, *e.g.*, to particular layers or parallelism settings, *i.e.*, they have *preconditions*. TRAINCHECK automatically infers training invariants as well as their preconditions and enforces them to detect silent errors.

Scope. Silent errors in DL training span a broad spectrum, ranging from correctness violations such as buggy API implementations to optimization-sensitive issues like hyperparameter choices. This work specifically focuses on correctness violations, which directly affect the integrity and correctness of training outcomes, and are often high-impact and actionable, yet remain largely overlooked by existing tools. Within this scope, TRAINCHECK aims to provide early and accurate *detection* of silent training errors before they silently propagate and accumulate. While its detection results can offer useful diagnosis hints to developers, providing systematic debugging support for precisely identifying root causes deserves dedicated investigation in future work.

3.1 Overview

Figure 3 shows the workflow of TRAINCHECK, which operates in two phases. In the offline phase, it automatically infers a set of training invariants from high-quality training pipelines. In the online phase, these invariants are deployed to a given training job to check for violations during training. TRAINCHECK reports invariant violations with contextual information to assist confirmation and investigation.

TRAINCHECK comprises three major components: (1) Instrumentor, which dynamically instruments DL training programs to collect runtime traces with low overhead; (2) Infer Engine, which analyzes these traces to infer training invariants and their preconditions automatically; and (3) Verifier, which continuously validates a training task against the invariants. When Instrumentor is used in the online stage, it performs selective instrumentation relevant to the inferred invariants.

Relation	Description
$Consistent(V_a, V_b)$	V_a and V_b should have the same values, while the values may change
$EventContain(E_a, E_b)$	E_b must happen in the duration of E_a
$APISequence(I_a, I_b, \ldots)$	I_a, I_b, ... must all occur and in the specified order
$APIArg(I_a, is_distinct)$	Ensures argument consistency or distinction in all calls to I_a
$APIOutput(I_a, bound_type)$	The output of I_a must meet certain attribute constraints

Table 2: Sample relations TRAINCHECK defines and supports.

In this section, we focus on Infer Engine. § 4 will describe the design of Instrumentor. Figure 4 shows a simplified example of the concrete training invariant with preconditions TRAINCHECK infers for the Bloom-176B training error.

3.2 Invariant Representation

TRAINCHECK focuses on capturing semantic rules related to DL training. Such rules can be expressed as logical relations involving key variables (e.g., model weights), and events (e.g., API invocations). Infer Engine defines a generic *relation interface* from which the specific relations can be implemented. An invariant is a relation instantiated with concrete variables and APIs that must satisfy the specified relationship.

Through analyzing silent training errors in practice, we identify and provide a set of representative relations often violated in these silent errors. Table 2 lists five relations TRAINCHECK supports. For example,

- **Consistent** relation establishes that two attributes of different variables should be equal despite how the value changes. This is effective for distributed training, where the model layers can be partitioned or duplicated across workers, and for model architectures that use shared parameters.
- **EventContain** relation establishes that when an API is invoked, a specified child event, such as another API call or a variable state change, occurs within that invocation. For example, calling `Optimizer.step` should always contain updates to the model parameters and the corresponding opera-

tions that perform those updates. This relation is effective for catching misconfigurations or unexpected arguments that otherwise lead to silent control-flow deviations.

- **APISequence** relation establishes (1) the group of APIs that should be called together and (2) the order in which they should be called. Users often forget to call certain APIs or call them in the wrong order, which can lead to silent errors. For example, in a few rookie mistakes collected from StackOverflow, users forgot to call `Optimizer.zero_grad` before `loss.backward`, leading to training instability.

These relations are deterministic and encode strict semantics, enabling precise and timely detection of silent errors. Besides the built-in relations, TRAINCHECK is extensible, so developers can easily add new relations into the framework.

Invariant inference in TRAINCHECK begins by instantiating each relation with concrete *descriptors*. A descriptor is a predicate that selects which trace records or events an invariant should examine. API descriptors specify the API's name and, optionally, the expected arguments or return values. Variable descriptors specify a variable's type and attribute name, along with an optional expected value or value change.

3.3 Trace Representation

TRAINCHECK collects execution traces from a DL training program by instrumenting it to emit relevant runtime information. A raw trace consists of a sequence of records capturing API entry and exit points, as well as variable states. Each trace record is annotated with a timestamp and thread ID. TRAINCHECK further extracts high-level *events* from raw trace records to represent semantically meaningful behaviors. For example, an *APICallEvent* represents a complete API invocation, aggregating its entry and exit records along with derived attributes such as execution duration and nested events. These high-level events form the foundation for invariant inference by providing structure to raw trace data.

TRAINCHECK additionally introduces **meta variables**, contextual attributes for a trace record. These include properties such as the training iteration number, distributed training ranks, and active context managers. Users can also define custom meta variables, such as pipeline stage (*e.g.*, initialization, training, or evaluation). These meta variables are essential for precondition inference (§ 3.5), enabling TRAINCHECK to generate invariants that are both precise and interpretable.

3.4 Invariant Inference

TRAINCHECK takes a hypothesis-based approach to infer training invariants from traces. As summarized in Algorithm 1, the inference workflow contains three main steps. (1) **Hypothesis generation**: The engine scans through all traces to instantiate a relation with potential concrete descriptors. (2) **Hypothesis validation**: For each hypothetical invariant, TRAINCHECK validates it against the trace and records entities that match the descriptors as passing/failing examples based on whether the relationship holds. (3) **Precondition deduction**: Try to deduce the distinction between passing/-failing examples. Such distinction, if found, will be used as the precondition for the property, and if not found, the hypothetical property is invalidated and dumped.

Algorithm 1: Invariant Inference

```
Input: traces, relation_pool
Output: all_invariants
all_invariants ← [];
foreach relation ∈ relation_pool do
    hypotheses ← [];
    foreach trace ∈ traces do
        hypotheses.extend(relation.GEN_HYPOS(trace));
    foreach hypo ∈ hypotheses do
        foreach trace ∈ traces do
            relation.COLLECT_EXAMPLES(trace, hypo);
    foreach hypo ∈ hypotheses do
        preconditions ← INFER_PRECONDITION(hypo);
        if preconditions ≠ null then
            hypo.invariant.preconditions ← preconditions;
            all_invariants.append(hypo.invariant);
return all_invariants;
```

Algorithm 2: Hypothesis Generation for `Consistent`

```
Input: trace
Output: Generated hypotheses
variables ← trace.get_all_variables();
foreach (var_1, var_2) ∈ Combinations(variables, 2) do
    foreach (attr_1, attr_2) ∈
      CartesianProduct(var_1.attrs, var_2.attrs) do
        if exists_value_match(attr_1.states, attr_2.states)
            hypo ←
              NEW_HYPO(relation ← ConsistentRelation,
                entities ← [VarDesc(var_1.type, attr_1.name),
                VarDesc(var_2.type, attr_2.name)]);
            yield hypo;
```

Each relation type implements the methods to generate and validate hypotheses for that relation. These methods are invoked in the generic inference loop. For the motivating example, the `Consistent` relation can be instantiated with two `torch.nn.Parameter` object's `data` attribute (Figure 4). Algorithm 2 shows how the hypotheses are generated for this relation. The other relations' hypotheses generation and validation are defined in a similar vein.

3.5 Preconditions

Semantics in deep learning are often context-sensitive and can only be applied to a specific subset of the training pipeline. For example, the parameter consistency invariant in distributed training should only be checked for the same model layer across workers within the same training iteration and only for the parameters replicated instead of partitioned across workers. For another example, the output tensor dtype usually depends on the input tensor dtype, but when an autocast context manager is active, the output tensor dtype should be the

autocast dtype instead of the input dtype.

TRAINCHECK thus defines *preconditions* for a training invariant. Preconditions provide various benefits: (1) confidence in the invariant's accuracy, (2) reduction in false positives, (3) debugging, as when the invariant is violated, the precondition can help explain which part of the pipeline is causing the error, and (4) reduced overhead by only performing the applicable checks. In addition, preconditions enable **transferable invariants**, as they provide a very clear context of when the invariant should be applied and thus can be applied to other pipelines that share the same context.

3.6 Deducing Preconditions

We design an algorithm to deduce the weakest yet safe preconditions for each invariant. A precondition is considered *safe* if it provides a clean separation: evaluating to `true` for *all* passing examples and `false` for *all* failing examples.

A precondition consists of one or more *conditions* (predicates) that compare a field's values across all trace records of a given example. TRAINCHECK supports four types: `CONSTANT`, where the field's value is identical in every record and must match a specific required value; `CONSISTENT`, where the field's value is identical in every record without a specific value constraint; `UNEQUAL`, where the field takes different values across records; and `EXIST`, where the field appears in every record.

The deduction algorithm first scans through the passing examples and produces conditions for each example. It then forms a *candidate precondition* by using the conditions found. For each example, the algorithm finds the conditions satisfied in all trace records of that example. The candidate precondition is then formed by taking the conjunction of the conditions satisfied in all passing examples. This conjunction is verified against the failing examples to see whether the precondition is safe; if so, the algorithm returns that precondition.

By restricting the candidate precondition to a conjunction of conditions that hold in all passing examples, the algorithm implicitly assumes the invariant applies to only a single scenario, which is often not the case. For instance, the parameter consistency invariant holds both across (1) data-parallel workers and (2) LayerNorm parameters on tensor-parallel workers.When the initial candidate precondition is unsafe, the algorithm attempts to split the passing examples into subgroups based on remaining non-overlapping conditions and performs inference on each subgroup. If no further splitting is possible, the algorithm terminates and reports an inference failure. If the preconditions inferred from these subgroups are safe and collectively cover all passing examples, they are combined disjunctively to form the final precondition.

Prune Irrelevant Conditions Numerous conditions can be inferred from the trace, but not all are relevant to the invariant. For example, in the parameter consistency invariant for distributed training, it is likely that all the parameters have the same `is_cuda` attribute. TRAINCHECK prunes irrelevant conditions during the safety verification process by removing conditions not violated in any failing examples. These conditions evaluate to true in all examples and are not discriminative. This approach is simple yet effective in removing a good amount of noisy conditions unrelated to the invariant.

A more complex situation is when the irrelevant condition is an artifact of the invariant handled by the current algorithm. For example, all consistent model weights will also have the same gradient values to maintain consistency. However, using consistent gradient values as a condition for the parameter consistency invariant is not a good idea because, while safe, it is too shallow and prevents the algorithm from going deeper to find the multiple scenarios that the invariant holds. To fix this problem, we allow each relation to encode rules on what conditions should be avoided in the precondition inference process. For example, a `Consistent` invariant, if about attributes of type `torch.Tensor`, cannot use any other attributes of type `torch.Tensor` as a condition. Alternatively, static analysis can help determine correlated fields and remove them from the precondition inference process, but its complexity and overhead may not be justified by the benefits.

Example In the simplified example presented in Figure 4, there is one passing example and two failing examples. Each example contains two trace records of `torch.nn.Parameter` objects. The algorithm first generates the candidate precondition by finding the conditions that hold in the positive example, which is `CONSTANT(tensor_model_parallel, False) && CONSTANT(is_cuda, True) && UNEQUAL(meta_vars.TP_RANK)`

Verifying the conjunction of these three conditions against the failing examples reveals that it is safe. However, the second condition of `is_cuda` constantly `True` is not violated in any failing examples; thus, it is pruned from the precondition. The final precondition is `CONSTANT(tensor_model_parallel, False) && UNEQUAL(meta_vars.TP_RANK)`

When the candidate precondition is unsafe, we are in an *under-constrained* situation. This can happen for three reasons: (1) the precondition is fundamentally not expressible in the condition types we support, (2) the information in the trace is not enough to infer the precondition, or (3) the invariant holds under multiple preconditions. In our experience, (3) happens most of the time. This indicates that local attributes and meta variables are sufficient for most invariants and do not need a complex grammar to express the preconditions.

We enhance an unsafe candidate precondition by adding conditions not considered previously to further constraint it, as described in § 3.6. We choose the conditions to add based on decreasing order of statistical significance, i.e., the conditions that cover the most passing examples. In Figure 5, the candidate precondition is the conjunction of the two fully-covering conditions, `cond1 && cond2`. However, since it is unsafe, we attempt to add `cond3` and `cond4`, resulting in a safe precondition of `cond1 && cond2 && (cond3 || cond4)`. If this precondition is still unsafe, we repeat the process until we find a safe precondition, or the computation budget is exhausted. The statistical significance based search helps reduce the search

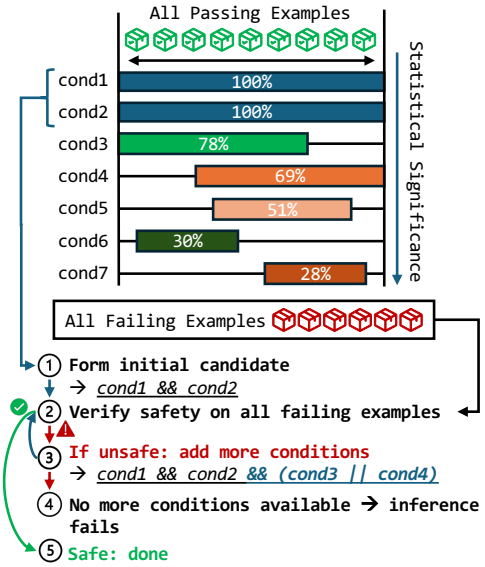


Figure 5: Deduce precondition in under-constrained situations.

space and find the majority scenarios that the invariant holds.

Our algorithm is not guaranteed to find strictly the weakest precondition, as the pruning strategy only looks at individual conditions, and we do not rely on any static program analysis to infer the preconditions. However, it provides a good trade-off between simplicity and effectiveness.

3.7 Filtering Out Superficial Invariants

One challenge in invariant inference is distinguishing meaningful invariants from superficial ones—those that appear valid due to the limited information available in a trace. For example, two irrelevant APIs `torch.cuda.is_available()` and `torch.jit.is_scripting()` can have consistent return values.

TRAINCHECK deems an invariant superficial if it cannot deduce a precondition for this invariant. Such invariants will not be deployed because they may not hold in the target training task. Even if they hold, they are not effective as runtime checks because we do not know when to apply them due to the lack of precondition. This is a key design choice to reduce false positives and enhance explainability.

Our approach differs significantly from traditional invariant mining solutions. They typically use the number of passing examples and the number of failing examples to estimate the statistical significance and select likely invariants. This approach has two drawbacks in our context. In DL training, having a large number of failing examples does not mean the invariant is superficial due to the diverse and intricate training process. Local semantics might not be statistically representative in the global context. For example, in tensor parallelism, all major parameters in the attention layer are partitioned across workers. Only the LayerNorm parameters are replicated, which accounts for less than 0.1% of the total parameters in representative transformer models like GPT [39]. During inference, the invariant that `torch.nn.Parameter` objects should be consistent has a passing/failing ratio of 1:38. This can be pruned by a statistical significance-based approach, but in reality, this turns out to be an important invariant for catching the Bloom-176B error.

3.8 Scalability

DL training produces large traces due to frequent API calls and variable updates, posing significant challenges for analysis. For example, instrumenting a 2-GPU, 70M-parameter pretraining run for BLOOM-176B (using Megatron-DeepSpeed's GPT-2 pipeline) generates approximately 92,000 trace records (50 MB) per training iteration.

TRAINCHECK adopts two key design choices to address this challenge. First, it restricts analysis to a predefined set of high-level semantic relations, significantly reducing the search space. Second, TRAINCHECK abstracts APIs and variables into *descriptors*. For variables, they are typically compound objects with multiple attributes. A variable descriptor consists of the variable's type (as returned by `type(obj)`), attribute name (*e.g.*, `data`, `grad`), and optional value constraints. This allows TRAINCHECK to reason over groups of variables sharing the same type and attribute, rather than enumerating individual instances. For example, when analyzing BLOOM-176B using a 2-GPU and 70M-parameter run, enumerating 104 variable instances such as `0.input_layernorm.weight` and `2.post_attention_layernorm.weight` would yield 5,356 pairs to consider. In contrast, TRAINCHECK considers only the available PyTorch variable types relevant to training state, where the primary type is `torch.nn.Parameter`. This dramatically reduces the number of invariant candidates while preserving the properties needed for error detection. Descriptors can also encode constraints (*e.g.*, requiring non-null values) to further generalize invariant specification.

3.9 Input Requirements

Importantly, while the inferred invariants are intended to validate large-scale training, generating them does not always require large-scale setups. In our experiments, effective invariants can often be inferred from small-scale runs. For example, although the original BLOOM training job spanned hundreds of GPUs, TRAINCHECK was able to infer the relevant invariant using only a 2-GPU run. All evaluated invariants in this paper were inferred from training jobs using at most 4 GPUs and 100 iterations. These results demonstrate that representative behaviors can be captured with lightweight workloads, making the inference phase practical and efficient.

4 Implementation

We implement TRAINCHECK in Python with 22.7 K lines of code. The system is composed of Instrumentor, Infer Engine, and Verifier. A key challenge in developing TRAINCHECK is to balance fine-grained trace collection for inferring effective invariants to detect silent errors and minimizing runtime overhead. We made extensive efforts to explore different techniques for achieving a good trade-off.

4.1 Instrumentor

Instrumentor collects traces from a training task for both Infer Engine and Verifier. It instruments a given DL training program to emit information as required by the relations we support (§ 3), specifically (1) *API invocation trace:* function entry, exit, arguments, and return values, (2) *variable state trace:* variable assignment, deletion, and modification, and (3) *meta variables:* steps, epochs, ranks, etc.

We implement it as a command-line tool that takes the path to the entry Python program and the libraries of interest (*e.g.*, `torch`, `deepspeed`) as arguments. Instrumentor automatically scans the `import` statements and model definitions in the program and instruments the necessary functions and variables. The user can also provide a shell script that sets up the environment and arguments for the program. Trace logs are written to a specified directory using JSON format.

Instrumentor is designed with three key goals: (1) non-intrusiveness, (2) low overhead, and (3) high coverage. Instrumentor also aims to be flexible in instrumentation granularity, as different relations require different levels of detail. For example, the `Consistent` relation only requires a periodic sampling of model states, whereas the `EventContain` relation requires an eager model state logging whenever the variable is modified, as accurate timing of events is crucial for it.

One option is to use Python's `sys.settrace` function, which installs a trace function that is invoked on every function call, return, and exception. However, this approach incurs prohibitively high overhead; we observed a 200× to 550× slowdown. It can also interfere with existing workflows that use tools relying on `sys.settrace`, such as `pdb` and `cProfile`.

Dynamic Monkey Patching To meet the above goals, we adopt a *monkey-patching* approach. Instrumentor is implemented as a Python package that dynamically instruments the target program by injecting hooks into relevant source code at runtime. It supports *selective* instrumentation, allowing users to specify which modules to instrument; only APIs defined in the specified modules are patched. During instrumentation, Instrumentor recursively traverses the namespace of each selected module and wraps identified methods. Each wrapper inserts logging and bookkeeping logic before and after invoking the original function. To minimize overhead, Instrumentor skips low-level internal functions, particularly those in `torch.jit` and `torch._C`, which are invoked frequently but rarely provide meaningful information for invariant inference.

Tracking Variables Instrumenting Python variables is more challenging than tracing APIs. In CPython, assignment operations occur at the C level and do not invoke any Python-level hooks, making it infeasible to efficiently track arbitrary variable state changes. Our study shows that most non-trivial silent errors, despite their diverse root causes, affect a small set of key objects, such as the model and optimizer. If a silent error does not impact model quality, it is often inconsequential for training correctness. This leads to a key insight: tracking the state of only the model and optimizer is sufficient to detect meaningful silent errors. These objects are typically long-lived, and updates to them occur through attribute modifications rather than object replacement, which simplifies tracking. Therefore, Instrumentor focuses on tracking models and optimizers rather than arbitrary local variables.

To achieve this, models and optimizers are wrapped with a `Proxy` that intercepts state-changing operations via overridden magic methods such as `__setattr__`. These changes are logged to the trace eagerly upon execution. During instrumentation, Instrumentor scans the program's AST to locate the initialization sites of these objects and replaces them with their corresponding `Proxy` instances. Alternatively, when precise timing of state changes is not required, Instrumentor supports a lower-overhead, sampling-based approach that registers a state-dump callback on `Optimizer.step`.

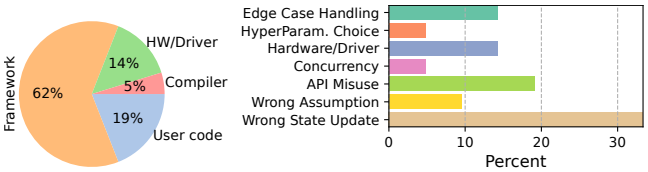
Logging Hashes of Tensors When dumping the model states, we are essentially doing checkpointing. The cost of this is unbearable as the model states are large and the overhead of serializing them, writing them to the log file, and reading them back is significant. Through our study, we observe that the actual values of the tensors are typically unimportant for inferring invariants. Silent errors typically arise from the shape, dtype, and the equality relationship between the tensors. Thus, Instrumentor only logs the hash of tensors.

Collecting Meta Variables Instrumentor also collects meta variables such as the current step, epoch, rank, *etc.* Whenever a trace record for a function call or variable state is dumped, Instrumentor walks through the call stack and finds the loop index local variable. This is a simple heuristic that works for most cases, as the loop index is usually a local variable in the outermost loop that is incremented in every iteration. Instrumentor further allows users to specify meta variables by calling the `set_meta` API. Users may also annotate the program into different phases, such as training, validation, and testing. The general idea for collecting meta variables is to provide context for the invariants to deduce preconditions.

4.2 Infer Engine

Infer Engine processes the trace files generated by Instrumentor to infer training invariants and their preconditions. One key challenge is handling large input traces produced by the data-intensive nature of DL training. Typical training pipelines can generate several hundred megabytes of trace data per epoch. Our algorithms, as described in § 3, are designed to efficiently deduce invariants with preconditions from large traces.

We implement Infer Engine in Python to leverage its rich ecosystem of libraries for data processing and deep learning. In addition, our traces cannot be conveniently represented with a fixed schema. We implement multiple trace backends, including Pandas, Polars, and built-in dictionaries. From experimenting with these backends, we choose Pandas DataFrames with Python dynamic typing as the default based



(a) Error locations. (b) Root cause types.

Figure 6: Root cause types of errors we reproduce for evaluation.

on its analysis performance and flexibility. We introduce optimizations with custom analysis functions that provide query caching, sampling, and pruning (*e.g.*, pruning candidates related to `torch.cuda.is_available`).

4.3 Verifier

During the online checking phase, TRAINCHECK consumes the stream of the trace generated by the instrumented program to detect silent errors in real-time. Different from the offline phase, the instrumentation is restrained to only the APIs and variables that are relevant to the deployed invariants and is thus lightweight. Verifier monitors the trace and triggers a check when a relevant piece of trace is available. It first evaluates whether the preconditions of an invariant are satisfied. If so, it checks whether the invariant holds. When an invariant violation is detected, Verifier reports the invariant and the corresponding trace to provide debugging help to developers.

5 Evaluation

We evaluate TRAINCHECK to answer several questions: (1) Can TRAINCHECK infer effective training invariants for detecting real silent errors? (2) How quickly can the invariants detect the errors? (3) Can TRAINCHECK help diagnose a detected silent training error? (4) Is the detection accurate? (5) What is the runtime overhead?

Our experiments are run on a server running Ubuntu 22.04, equipped with an Intel(R) Xeon(R) Silver 4310 CPU, 252 GB RAM, and one NVIDIA A40 GPU. For trace collection related to distributed training, we use another server with identical specifications but featuring 8 NVIDIA A2 GPUs. We use Python 3.10 and PyTorch 2.2.2 with CUDA 12.1.

5.1 Silent Error Detection

To assess TRAINCHECK's effectiveness, we collect and reproduce 20 *real-world* silent errors. Of these, 6 are drawn from our prior study (§ 2), while the remaining 14 are newly gathered from GitHub issues for popular libraries (PyTorch, DeepSpeed, etc.), StackOverflow, and social media. These errors span a broad range of root cause locations and types, as illustrated in Figure 6a and Figure 6b. The complete descriptions of these errors are available in our technical report [23].

We compare TRAINCHECK with four baselines that represent the current practices and state-of-the-art research:

- *Spike* detector is used to monitor numerical instability in training where the loss or accuracy spikes to a large value.
- *Trend* detector is used to monitor the training process where the loss or accuracy is not decreasing or increasing as expected. A tolerance factor is set to allow some fluctuation.
- *Anomaly Detection* detector applies common algorithms like LOF, Isolation Forest, and Z Score to detect anomalies on the same high-level metrics like loss and accuracy.
- *PyTea [22]* and *NeuRI [28]* are two recent research artifacts. PyTea specifies constraints on APIs used in the training process, primarily focusing on the shaping constraint of the input and output tensors. NeuRI automatically infers such constraints encoded in PyTea's syntax.

When implementing the first three detectors, we monitor signals as per industry practice [5] and apply the same configuration parameters to all errors for a fair and consistent comparison. For the spike detector, we set the threshold to 75, and for the trend detector, we set the tolerance to 3. The number of neighbors for LOF is set to 2, and the contamination factor is set to 0.1 for Isolation Forest. Other parameters use defaults provided by Scipy and Scikit-learn.

Methodology We prepare a reproduction script for each error and run them to emit (1) runtime trace for the checking of TRAINCHECK and PyTea/NeuRI, and (2) high-level metrics (loss, accuracy, gradient norm) for the anomaly detection-based detectors. We then run the detectors on the traces and metrics and collect the detection results. The invariants used by TRAINCHECK are inferred from PyTorch's official GCN, Autocast, and DDP examples, for PyTorch-related errors, and Megatron-DeepSpeed's official GPT pretraining examples for DeepSpeed-specific errors, and the official Transformers trainer examples for Transformers-specific errors.

We focus on *true detections* (true positives) to avoid rewarding detectors that indiscriminately raise many alarms. This ensures the result reflects real error-detection capability. For example, we observe that when using the anomaly detection detector, no matter how we tune it, it raises numerous alarms throughout the training process, *e.g.*, since the loss is dropping fast. To objectively determine the true positive, we run the fixed versions of each error and check if the detector also raises alarms in the error-free traces.

Detection TRAINCHECK successfully detects 18 out of the 20 errors. The invariants it infers and uses represent all five relations in Table 2. In all cases, detection occurs no later than one iteration after the root cause is triggered. For the motivating example, the incorrect gradient clipping logic is triggered in the second training iteration, and TRAINCHECK shortly detects it in the third iteration. Despite diverse root causes, TRAINCHECK achieves high detection coverage and timeliness. We attribute its effectiveness to our invariant checking approach and the precision of the inferred invariants.

TRAINCHECK fails to detect two errors: TF-33455 and TF-29903. TF-33455 involves the trainer stopping early due to an incorrectly calculated total number of training steps, while the training process itself is correct. Detecting this error would re-

Bug Id	Synopsis
AC-2665	Initializing the optimizer prior to wrapping the model with DDP causes training to not progress.
DS-6770	A mismatch between the model and the parameters held by the optimizer causes a KeyError during initialization.
DS-5489	Freezing parameters prior to initializing DeepSpeed causes incomplete model checkpoints.
DS-6714	Using heterogeneous MoE architecture with pipeline parallelism causes inconsistent usage of communication primitives, leading to training stuck.
DS-6772	DeepSpeed initialization silently overwrites "id" attributes on models, causing wrong model-GPU placement.
DS-6089	The program is stuck on communication due to consistent "capacity" value across workers.

Table 3: Six newly reported bugs that lead to silent errors, detected and diagnosed with the help of TRAINCHECK. AC: Accelerate. DS: DeepSpeed. Numbers refer to GitHub issue IDs.

quire monitoring the computed training steps and comparing them with the intended arguments. Currently, TRAINCHECK does not support tracking Python primitive variables, as doing so would incur prohibitive overhead and require modifications to the Python runtime. TF-29903 concerns a bug caused by a corrupted state dict constructed within the `safe_checkpoint` function. TRAINCHECK fails to detect this case because (1) this error is confined to the checkpoint function and does not impact the main training logic, and (2) TRAINCHECK does not analyze local variables.

In comparison, the signal-based detectors collectively only detect 2 errors, which are extreme cases where the model stops learning entirely and the loss is constant over epochs. The PyTea/NeuRI detector detects 1 error, which is from a bug in the transformers library where the processed data does not have the same batch size as the argument, falling into the shaping constraints supported by PyTea/NeuRI.

Diagnosis While diagnosis is not the primary goal of TRAINCHECK, we conduct analysis to understand whether invariant violations can aid in debugging. Among the 18 cases detected by TRAINCHECK, the violation reports can pinpoint the exact root cause in 10 cases and localize close to the root causes in 8 cases. The diagnosis hints for the one error detected by PyTea/NeuRI are on par with TRAINCHECK.

5.2 New Silent Errors

To further test TRAINCHECK's effectiveness, we monitor recent open GitHub issues in DeepSpeed and Transformers. We focus on issues that have silent symptoms, are unresolved with unknown root causes, and reproducible. We create reproduction scripts based on the reports to ensure the issues occur with the latest library. We then apply the invariants TRAINCHECK infers from the sample pipelines to the issues.

During this exercise, TRAINCHECK detects 6 new silent errors at an early stage and aid in diagnosing their root causes, as summarized in Table 3. Three of these root causes have since been confirmed and fixed.

Case Study: AC-2665 It causes the model to not learn effectively during training. In the original issue report, the user adapted their original pipeline into DDP, but the model stopped learning at all. The user did not know the root cause, but found a setting that fixed the error, `use_orig_param true`.

We applied the invariants inferred from the GCN example to the user's pipeline and inspected invariant violations. Below, we present three example true positives identified by the inferred invariants. Inv1: `zero_grad` should contain changing of `grad` attributes from a non-zero tensor to a zero tensor or None. Inv2: `step` should contain changing of `data` attributes of the model parameters. Inv3: `step` should contain a number of invocations of mathematical operations on the model parameters, such as `_foreach_add`. Inv2 and Inv3 indicate that the optimizer is not performing updates to the model, while Inv1 indicates that no gradient was computed. The three invariants combined point out that the optimizer is likely not initialized with parameters that are actually used during forward and backward passes. We then checked the model parameters and the optimizer's param_groups and confirmed the hypothesis. Upon investigation of the model and optimizer initialization process, we found out that DDP automatically flattens the original model parameters and creates a new model with the flattened parameters. However, the user initialized the optimizer with the original model parameters, and thus the optimizer did not have the correct parameters to update. We reported the issue both to the user and to the transformers team, and the issue has been confirmed. A pull request has been under review to fix the issue.

5.3 False Positive

To measure false positive, we collect 63 diverse training programs drawn from existing tutorials, all without known bugs. They span different training scales (*e.g.*, single-GPU vs. multi-GPU), frameworks (PyTorch, Transformers, Diffusers, *etc.*), tasks (image classification, language modeling, vision transformer pretraining, *etc.*), and configuration parameters (*e.g.*, precision, batch size, dataset, and model architecture).

To minimize confounding effects from applying invariants across unrelated tasks, we group programs into four classes based on training task type, as shown in Figure 7. Transferability across task classes is evaluated separately in § 5.4. For each class, we split the programs into a training set (used to infer invariants) and a validation set (used to assess false positives). Validation programs are further categorized as either *cross-configuration* (differing from the training set only in configuration parameters) or *cross-pipeline* (different code with similar semantics). This categorization allows us to assess how well the inferred invariants generalize across both minor and structural variations in training programs.

Figure 7 shows that TRAINCHECK achieves consistently low false positive rates. In the primary evaluation setting, where invariants are inferred from representative workloads using 5 or 6 input programs, false positive rates remain below

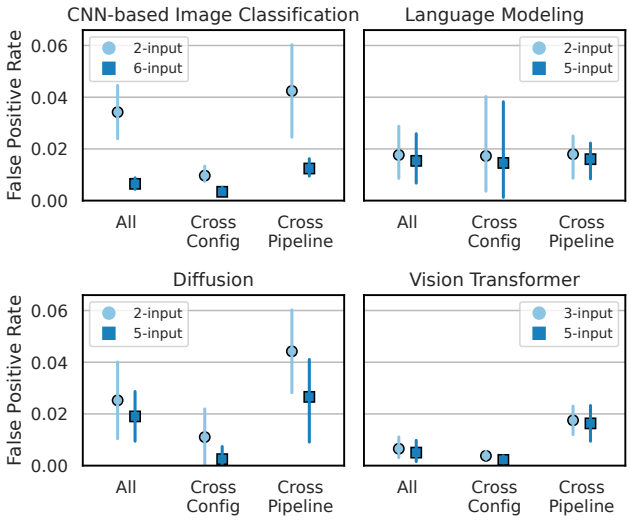


Figure 7: False positive rates across four program classes, broken down by cross-configuration and cross-pipeline settings.

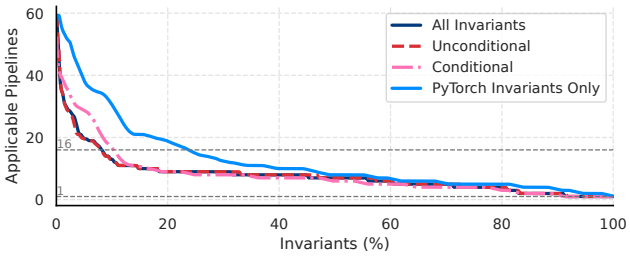


Figure 8: Invariant applicability across all collected pipelines. The y-axis shows how many pipelines each invariant applies to. "All Invariants" shows overall coverage; "PyTorch Invariants Only" isolates PyTorch-specific APIs. "Conditional" and "Unconditional" indicate whether preconditions are present.

2% across all classes. Even in constrained settings with only 2 or 3 input programs, the rate stays below 5%.

5.4 Invariant Transferability

TRAINCHECK infers transferable invariants, enabling those learned from a small set of input programs to generalize across different pipelines and library versions. To evaluate this transferability, we apply a set of *valid* invariants to all 63 collected pipelines. These invariants are inferred using a 5/6-input setup across all classes and exclude any that triggered false positives in § 5.3. For each invariant, we count how many pipelines it can be applied to without raising a false alarm.

As Figure 8 shows, many invariants exhibit broad transferability. All invariants apply to at least one additional pipeline beyond those used for inference. Notably, over 8% of invariants apply to more than 16 pipelines—the average number of pipelines per model class, demonstrating strong cross-class generalization despite semantic and structural differences. We also observe that invariants with preconditions are generally more transferable than unconditional ones, underscoring the importance of precise precondition inference.

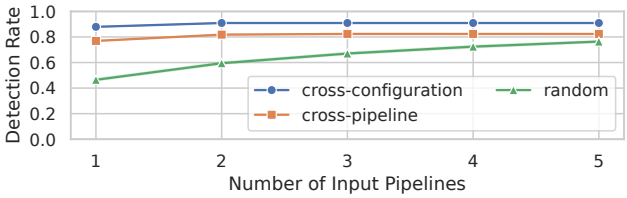


Figure 9: Detection rate as the number of input pipelines increases under cross-configuration, cross-pipeline, and random settings.

We expect the transferability to be even higher in practice, as developers typically apply invariants only within the same framework or library context. For example, invariants involving Transformers-specific APIs would not be applied to a program that only uses PyTorch. Since all pipelines in our dataset use PyTorch, we isolate 8,172 invariants that capture only PyTorch-specific semantics. These exhibit significantly higher transferability: 23% of them apply to more than 16 pipelines. This result suggests that framework-level behavior is a strong source of reusable invariants.

We also evaluate the false positive rate of applying invariants across model classes to assess TRAINCHECK's robustness in extreme transfer scenarios. Using the setup from § 5.3, we infer invariants from one class and apply them to all programs in the other classes, comparing the resulting false positive rate to the in-class baseline. Surprisingly, only the smallest-scale class, CNN, shows a higher false positive rate in the cross-class setting (2.62% vs. 0.65%). In all other cases, the rate is comparable or lower (*e.g.*, 0.93% vs. 1.54% for language modelling). This is because many invariants become inapplicable due to differing API usage and training context.

5.5 False Negative

To study the trade-off between false negative and the input programs used for invariant inference, we evaluate three settings: *cross-configuration*, *cross-pipeline*, and *random*. In the *cross-configuration* setting, invariants are inferred from historical runs of the same training pipeline executed under alternative configurations where the silent error was not observed. In the *cross-pipeline* setting, invariants are inferred from semantically similar training pipelines that do not exhibit the error. In the *random* setting, invariants are inferred from general tutorial pipelines collected from relevant frameworks.

For each silent error detected in § 5.1, we randomly sample k input pipelines for each setting and evaluate whether the error can be detected using the inferred invariants. The average detection rate for a given k is computed by repeating this process 100 times and averaging the results across all cases.

As Figure 9 shows, increasing the number of input pipelines consistently improves detection rates. Both the cross-configuration and cross-pipeline settings achieve high detection coverage (91% and 82%, respectively) even with only two input programs. The random setting starts with lower detection coverage, but improves steadily as more inputs are added (76% with five inputs). Further investigation of the

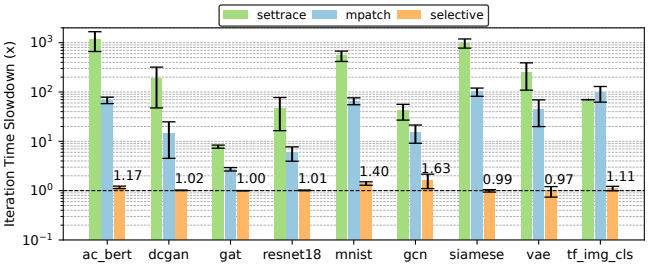


Figure 10: Overhead of different instrumentation techniques.

undetected silent errors reveals that the violated semantics often involve specialized features that are underrepresented in the available example pipelines or not exercised in cross-configuration or cross-pipeline inputs. For instance, detecting `DeepSpeed-5794` requires invariants about DeepSpeed's MoE features; however, only 1 out of 15 available DeepSpeed tutorial pipelines performs MoE training, making it infeasible to detect `DeepSpeed-5794` with random sampling.

5.6 Overhead

We evaluate the runtime overhead of invariant checking during training. TRAINCHECK uses *selective instrumentation*, which only instruments the APIs and variables relevant to the deployed invariants. We conduct experiments on a diverse set of training programs that vary in model size, task complexity, and framework usage. For each program, we deploy 100 randomly sampled invariants and compare the per-iteration training time before and after instrumentation. We compare our selective instrumentation against two baselines: (1) Python's *sys.settrace*, using a simple trace function that logs API calls and arguments without variable tracking, and (2) TRAINCHECK's *full instrumentation* mode, which instruments all API calls and variables regardless of invariant relevance.

Figure 10 shows the results. TRAINCHECK incurs low overhead in selective mode, typically less than 2%, and at most 1.6× slowdown across all workloads. `GCN` and `MNIST` exhibit higher relative overhead (1.6× and 1.4×, respectively), as they are toy workloads (*e.g.*, training a 2-layer CNN) where per-iteration execution time is minimal, and any instrumentation incurs a larger proportional cost. In contrast, for the more realistic workloads, overhead is significantly lower, as a larger portion of the time is spent on GPU-bound computation.

The primary sources of overhead stem from trace data serialization into JSON, conversion of objects into dictionary representations, and handling of tracked objects. These operations occur at the Python level and introduce runtime costs, especially in tight CPU-side loops. While our current implementation is synchronous and prioritizes correctness and modularity, there are clear opportunities for reducing overhead further. Potential engineering optimizations include asynchronous or batched logging and minimizing redundant instrumentation through static analysis. We leave the exploration of these optimizations to future work.

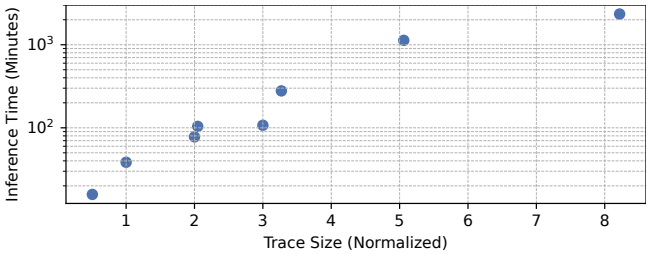


Figure 11: Inference time vs. trace size (normalized to ResNet-18 pretraining trace size).

5.7 Inference Efficiency

To quantify the invariant inference efficiency, we reuse the traces from the false positive experiments (§ 5.3) and measure the inference time across traces of varying size and structural complexity. For consistency, we normalize trace size by treating the trace from a ResNet-18 pretraining run with 20 training iterations and 10 testing iterations as a standard program trace, corresponding to 66.2 MB or 93,686 records.

Figure 11 shows that inference time grows roughly quadratically with trace size in this setting. While the inference algorithm itself is linear in both trace size and the number of hypotheses, larger traces typically expose more semantic behaviors, resulting in a larger hypothesis set. In the worst case, TRAINCHECK completes inference in 38 hours when processing traces from 8.2 standard programs. Since inference runs offline and our current implementation is single-threaded, the performance remains acceptable for practical usage.

5.8 Examining Invariant Violations

Despite a low false positive rate, false alarms do occur, and developers must examine violation reports to extract actionable diagnostic information. We find that violations can often be inspected structurally rather than treated in isolation. They tend to cluster around specific APIs or components, which makes the review process more manageable. In practice, true positives are frequently supported by multiple related violations that reinforce the underlying issue, while false positives follow recognizable patterns and are easy to dismiss.

For example, in the AC-2665 case study (§ 5.2), using invariants inferred from PyTorch's GCN pipeline alone, 100 violations were reported. Among these, 52 were true positives. Of those, 33 indicated that `torch.optim.adamw.adamw` was never invoked, highlighting a missing optimizer initialization. 18 showed that `optimizer.step` performed no mathematical operations, suggesting it was not linked to model parameters. The remaining 48 violations were quickly dismissed as irrelevant: 7 involved GCN-specific constants (e.g., `dropout_rate == 0.5`), and 26 flagged missing ReLU invocations, which do not apply to the T5-based model used in AC-2665. The rest followed similar, non-impactful patterns.

6 Limitations

TRAINCHECK has a few limitations. First, its instrumentation interferes with JIT compilation tools like `torch.compile`, preventing the analysis of optimized code paths. Second, it is restricted to Python code, limiting its ability to analyze components with significant logic implemented in lower-level languages, such as the Flash Attention algorithm [11]. Finally, representing tensors in hash form prevents fine-grained numerical analysis, limiting its applicability to detecting instabilities caused by inappropriate hyperparameters. However, this can be complemented by existing research on hyperparameter tuning [50] and numerical defect detection [25].

7 Related Work

Testing Pipeline Code, Libraries, and Compilers CRADLE [33], AUDEE [18], LEMON [44], and NNSmith [27] employ differential testing to detect and diagnose failures across frameworks and compilers. PyTea [22] detects tensor shape mismatches using pre-specified API constraints, while NeuRI [28] enhances this by automating constraint inference. RANUM [25] targets numerical defects in deep neural networks. TRAINCHECK goes beyond framework inconsistencies or isolated error categories by inferring runtime invariants tailored to the training process. It automates error detection and debugging with automated invariant inference from example pipelines, addressing a broader range of silent failures across diverse training pipelines.

Monitoring Frameworks for Training Dynamics Tools like TensorBoard [1] and Weights & Biases [5] log high-level metrics such as loss and accuracy, enabling developers to visualize and compare experiments easily. However, these tools require manual and active monitoring, as seen during the training of BloombergGPT [45], where loss plateaued for seven days before the developer noticed it. These metrics are often noisy and can lead to many false alarms when used for detecting silent errors. They also do not help with diagnosis.

In contrast, TRAINCHECK provides an automated solution that captures the precise semantics of training as training invariants and proactively checks them, enabling reduced manual effort, early and accurate detection of silent training errors, and providing diagnosis hints for root-cause analysis.

Testing DL Models Extensive work exists to test trained DL models for robustness and fairness, such as DeepXplore [32] and DeepTest [43]. They are orthogonal to TRAINCHECK given their focus on testing and uncovering errors in the final model weights rather than validating the training process.

Fault-Tolerance in DL Systems Fault-tolerance mechanisms have been proposed, such as elastic resource scaling, task reallocation, pipeline parallelism, and efficient checkpointing [3, 21, 26, 42]. While these approaches improve fault tolerance, they do not address silent errors that arise from misconfigurations, bugs, or subtle correctness violations.

Invariant Mining Much work has explored mining likely program invariants for traditional single-component software with tools such as Daikon [16] and DIDUCE [19]. These invariants focus on low-level program variable relations at certain points of the program, such as `idx < len` for two local variables in a loop. Recent work such as I4 [30], DistAI [49], and DuoAI [48] infer inductive invariants, which are used in the verification of distributed protocols. Oathkeeper [29] infers event rules to detect silent failures in distributed systems.

Inferring rules is a general approach for bug detection. Engler *et al.* [15] notably propose inferring rules about programmer beliefs and show its effectiveness in large systems code.

TRAINCHECK targets a new domain of deep learning training systems and addresses various challenges unique to this domain. To the best of our knowledge, it is the first work for systematic invariant checking in DL training pipelines to detect silent training errors. The training invariants TRAINCHECK infers capture high-level semantics tailored to DL training. TRAINCHECK also deduces precise preconditions for these invariants. Moreover, prior work mainly infers invariants from a single system, and these invariants only apply to that system. In contrast, TRAINCHECK invariants from diverse and seemingly unrelated training pipelines. Its inferred invariants are transferable to different pipelines.

8 Conclusion

Silent errors are detrimental to DL training and yet notoriously difficult to address. This paper presents a study on such errors to understand their characteristics. Informed by the study, we propose a principle approach that uses precise training invariants to detect and diagnose silent training errors. We present TRAINCHECK, an end-to-end framework that automates the process of inferring training invariants and proactively checking them. TRAINCHECK shows effective and quick detection capability for real-world silent training errors with diverse root causes. It also uncovers previously unknown bugs leading to silent errors in popular training libraries. Its tailored approach and the transferability of its inferred invariants make it readily applicable to existing DL training pipelines.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their constructive feedback and guidance, which greatly improved the quality of the paper. We are grateful to Stas Bekman for sharing detailed insights into the Bloom-176B training issue. We thank Yijun Wang for his contribution to the empirical study. We appreciate the feedback from the members of OrderLab and Mosharaf Chowdhury. We thank the Chameleon Cloud for providing experiment machine resources. This work was supported in part by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, et al. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, page 265–283, Savannah, GA, USA, November 2016. USENIX Association.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, La Jolla, CA, USA, April 2024. ACM.

[3] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, Rennes, France, April 2022. Association for Computing Machinery.

[4] Stas Bekman. BLOOM: Megatron-DeepSpeed. https://huggingface.co/blog/bloom-megatron-deepspeed, 2022.

[5] Lukas Biewald. Weights and Biases. https://www.wandb.com/, 2020.

[6] BigScience Workshop. Chronicles of BigScience TR11-176B-ML Training. https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md#2022-04-30-hanging-at-eval, 2022.

[7] Rishi Bommasani. Dropout with Theano. https://rishy.github.io/ml/2016/10/12/dropout-with-theano/, 2016.

[8] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, and et al. On the opportunities and risks of foundation models, 2022.

[9] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. Toward understanding deep learning framework bugs. *ACM Trans. Softw. Eng. Methodol.*, 32(6), September 2023.

[10] CodeParrot. CodeParrot Clean Train Dataset, 2021.

[11] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, page 16344–16359, New Orleans, LA, USA, November 2022. Curran Associates Inc.

[12] DeepSpeed Contributors. DeepSpeed GitHub Issues. https://github.com/microsoft/DeepSpeed/issues, 2024.

[13] Malinda Dilhara, Ameya Ketkar, and Danny Dig. Understanding software-2.0: A study of machine learning library usage and evolution. *ACM Trans. Softw. Eng. Methodol.*, 30(4), July 2021.

[14] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, et al. The Llama 3 Herd of Models, 2024.

[15] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 57–72, Banff, Alberta, Canada, 2001. Association for Computing Machinery.

[16] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 213–224, Los Angeles, California, USA, May 1999. Association for Computing Machinery.

[17] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature Medicine*, 25(1):24–29, January 2019.

[18] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 486–498, Virtual Event, Australia, December 2020. ACM.

[19] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 291–301, Orlando, Florida, May 2002. Association for Computing Machinery.

[20] Monica Heddneck. Dataloader not randomly sampling in PyTorch. https://stackoverflow.com/questions/50124712/dataloader-not-randomly-sampling-in-pytorch, 2018.

[21] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, Koblenz, Germany, October 2023. Association for Computing Machinery.

[22] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. A static analyzer for detecting tensor shape errors in deep neural network training code. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 337–338, Pittsburgh, Pennsylvania, May 2022. Association for Computing Machinery.

[23] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. Training with Confidence: Catching Silent Errors in Deep Learning Training with Automated Proactive Checks (Technical Report). Technical report, University of Michigan, 2025. Technical Report.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[25] Linyi Li, Yuhao Zhang, Luyao Ren, Yingfei Xiong, and Tao Xie. Reliability Assurance for Deep Neural Network Architectures against Numerical Defects. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1827–1839. IEEE Press, May 2023.

[26] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal Checkpointing: Efficient and Flexible Checkpointing for Large Scale Distributed Training, 2024.

[27] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '23, page 530–543, Vancouver, BC, Canada, March 2023. Association for Computing Machinery.

[28] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. NeuRI: Diversifying DNN Generation via Inductive Rule Inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '23, page 657–669, New York, NY, USA, November 2023. Association for Computing Machinery.

[29] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107, Carlsbad, CA, July 2022. USENIX Association.

[30] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 370–384, Huntsville, Ontario, Canada, October 2019. Association for Computing Machinery.

[31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, page 1–15, St. Louis, Missouri, November 2021. Association for Computing Machinery.

[32] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–145, Shanghai, China, October 2017. ACM.

[33] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, ICSE '19, pages 1027–1038, Montreal, QC, Canada, May 2019. IEEE.

[34] PyTorch Community. PyTorch Discussion Forum. https://discuss.pytorch.org/, 2024.

[35] PyTorch Contributors. PyTorch Examples. https://github.com/pytorch/examples, 2024.

[36] PyTorch Contributors. PyTorch GitHub Issues. https://github.com/pytorch/pytorch/issues, 2024.

[37] Tanel Pärnamaa. A Bug That Plagues Thousands of Open-Source ML Projects. https://tanelp.github.io/posts/a-bug-that-plagues-thousands-of-open-source-ml-projects/, 2019.

[38] Tanel Pärnamaa. Using PyTorch & Numpy: A Bug That Plagues Thousands of Open-Source ML Projects. https://www.reddit.com/r/MachineLearning/comments/mocpgj/p_using_pytorch_numpy_a_bug_that_plagues/, 2021.

[39] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. OpenAI, 2019.

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, Virtual Event, CA, USA, August 2020. Association for Computing Machinery.

[41] StackOverflow Contributors. StackOverflow - Questions and Answers on PyTorch. https://stackoverflow.com/questions/tagged/pytorch, 2024.

[42] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, pages 497–513, Boston, MA, April 2023. USENIX Association.

[43] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 303–314, Gothenburg, Sweden, June 2018. Association for Computing Machinery.

[44] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 788–799, Virtual Event, USA, November 2020. Association for Computing Machinery.

[45] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. BloombergGPT: A Large Language Model for Finance, 2023.

[46] Yuxin Wu. Unawareness of Deep Learning Mistakes. https://ppwwyyxx.com/blog/2017/Unawareness-Of-Deep-Learning-Mistakes/, 2017.

[47] Yuxin Wu. Fight Against Silent Bugs in Deep Learning Libraries. https://ppwwyyxx.com/blog/2020/Fight-Against-Silent-Bugs-in-Deep-Learning-Libraries/, 2020.

[48] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 485–501, Carlsbad, CA, July 2022. USENIX Association.

[49] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, pages 405–421, Virtual Event, USA, July 2021. USENIX Association.

[50] Tong Yu and Hong Zhu. Hyper-Parameter Optimization: A Review of Algorithms and Applications, 2020.

[51] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1159–1170, Seoul, South Korea, May 2020. Association for Computing Machinery.

[52] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, 2022.

[53] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering*, ISSRE '19, pages 104–115, Berlin, Germany, October 2019. IEEE.



NEUTRINO: Fine-grained GPU Kernel Profiling via Programmable Probing

Songlin Huang
The University of Hong Kong
huangs0@connect.hku.hk

Chenshu Wu
The University of Hong Kong
chenshu@cs.hku.hk

Abstract

As GPUs play an increasingly important role in computer systems in the scaling laws era, understanding fine-grained GPU runtime behavior is more crucial than ever. However, existing GPU kernel profilers, typically kernel-exclusive or hardware-dependent, often fail to capture fine-grained measurements. This paper presents NEUTRINO, a programmable interface for GPU kernel profiling that leverages assembly-layer probing to achieve instruction-level fine granularity, profiling versatility across time and value domains, and hardware independence. To better visualize the rich details captured by NEUTRINO, we introduce the Densified Memory Access Timeline (DMAT), a novel representation that offers new insights into GPU runtime behavior. We implement NEUTRINO in Linux for both NVIDIA and AMD GPUs and conduct extensive evaluations and analyses. The results demonstrate NEUTRINO's superior capabilities in GPU kernel profiling with low overhead. We envision NEUTRINO as a valuable tool for the community and have open-sourced it to facilitate future research at https://github.com/open-neutrino/neutrino.

1 Introduction

With the emergence of artificial intelligence under the scaling laws [41] in data, parameters, and computation, the underlying computer systems have been scaling rapidly, primarily driven by GPUs, parallelism-oriented computing devices that are heterogeneous from the traditional OS running on CPUs. The huge system scale and the unique parallelism-oriented design open many research challenges on GPU systems, such as communication [16, 36, 67, 70], memory efficiency [43, 78], computation pipelining [35, 46, 63, 65, 75], and GPU cluster scheduling [3, 37, 47, 52, 99]. To address these challenges, researchers would greatly benefit from comprehensive measurements and in-depth insights into the runtime behavior of real workloads on GPU systems. Demystifying the otherwise opaque GPU programs can promise new opportunities for optimizing machine learning (ML) systems.

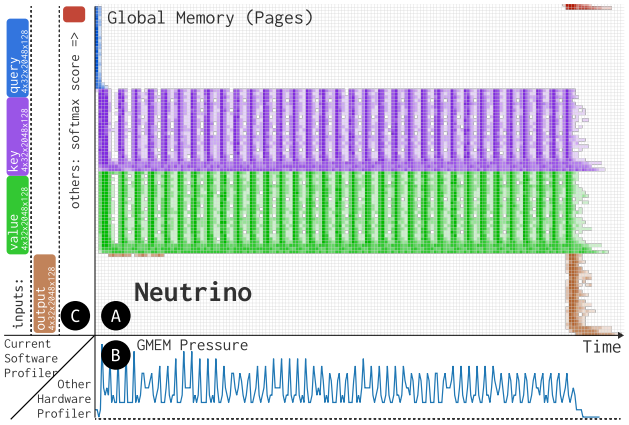


Figure 1: NEUTRINO's Densified Memory Access Timeline Plot. ❶ NEUTRINO expands new dimensions from previous ❷ hardware-dependent profilers and ❸ kernel-exclusive software profilers. Color depth in ❶ shows density from parallel threads, profiled from Flash-Attn-v2 [19] (non-causal).

However, profiling real GPU workloads at a fine granularity has been well-recognized as a significant challenge from previous attempts [5, 7, 23, 55, 58, 93]: ❶ The proprietary and heterogeneous hardware, coupled with the huge system scales (*e.g.*, 10,000+ cores), limits the capability to probe fine-grained information. ❷ GPU kernels are treated as atomic to the host OS, which largely prevents profiling GPU kernels through the mature OS profiling techniques [24]. ❸ Many profilers [12, 15, 18, 50, 74] rely on concurrency mechanisms like timer interrupts and locks, which are either unsupported or inefficient on GPUs due to the parallelism-oriented architecture. These challenges are further amplified by the rapid development of GPU systems with many new features introduced over the past few decades, such as matrix core [1, 73] in execution model and asynchronous copy [2] for memory access, which continuously introduce new runtime behaviors, performance issues, and profiling needs.

These unique challenges make existing GPU profilers ei-

ther *kernel-exclusive* [87], only capturing coarse-grained metrics like FLOP/s, or *hardware-dependent* [5,7,55,58], relying on physical hardware features such as Performance Monitor (PM) counters, as shown in Tab. 1. Additionally, these hardware profilers are *sampling-based*: They use hardware counter readings at certain intervals to capture statistics such as memory throughput, and cannot support more informative profiles like the page reference map [22] for capturing the spatial and temporal patterns of memory access. There are also explorations on GPU instrumentation [14, 77, 80, 82], such as NvBit [93] that manipulates proprietary machine code or HIPAnalyzer [21] that instruments compilers [44]. Yet these efforts still only focus on statistics such as memory access divergence among threads or reusable distance. To the best of our knowledge, there exist no tools for a fine-grained and general-purpose programmable interface of GPU profiling, just like eBPF [24] for Linux Kernel tracing.

To bridge this gap, we present NEUTRINO, a GPU assembly probing tool for *fine-grained*, *versatile*, and *programmable* GPU kernel runtime profiling. Inspired by eBPF [24], NEUTRINO's design aims to attach small snippets (probes) to GPU programs to expose runtime details of program executions. Specifically, NEUTRINO extracts, instruments, and reassembles GPU assemblies [4,30,60], rather than machine code [57] or compilers [44,45], allowing fine-granularity, versatility, and programmability in one framework:

■ **Fine-granularity:** NEUTRINO directly works on assemblies, the lowest software level, to offer the finest granularity at the instruction level that can be effectively mapped to particular hardware units such as tensor cores and memory I/Os.

■ **Versatility:** NEUTRINO supports GPU kernel profiling from both perspectives of *value*, *i.e.*, capturing runtime values such as memory addresses and of *time*, *i.e.*, recording event timestamps or even intra-kernel micro-benchmarking by differencing timestamps. By covering these two dimensions, NEUTRINO supports versatile profiling tasks from warp/block scheduling to memory access patterns.

■ **Programmability:** NEUTRINO extends the programmability of previous GPU instrumentation frameworks [14, 21, 77, 80, 82, 93] to cooperative probes by leveraging registers as the temporal storage between probes. By doing so, NEUTRINO enables more complicated and flexible profiling tasks by cooperating probes at different tracepoints and time.

NEUTRINO excels with its distinct probe design (§3) of three key components, namely snippet, tracepoint, and structured map, corresponding to the probe's target functionality, injection point, and output format, respectively. At runtime, NEUTRINO probes will be injected into tracepoints of the original program, and snippets use logically independent registers to place temporal results. This design, together with the GPU SIMT model, ensures the probes are virtual to the original program. Moreover, with eBPF-like structured maps, NEUTRINO probes can flexibly store metrics to one or more buffers via race-free saving without costly metadata.

We fully implement (§4) NEUTRINO for NVIDIA GPUs with the CUDA driver and AMD GPUs with the ROCm driver on Linux, consisting of three modules: the DSL compiler, the hook driver, and the probe engine. The DSL compiler compiles probes written in platform-independent Python Tracing DSL into raw low-level assembly probes wrapped in TOML [68]. The hook driver emulates symbolic links to the driver (shared library) to provide runtime support, including capturing GPU calls from the user, allocating probe maps, and saving results to the storage. The core probe engine validates, instruments, and reassembles the probed assembly code from wrapped low-level probes. Finally, NEUTRINO is encapsulated into an easy-to-use CLI similar to `bpftrace` [13] that can be run by `neutrino -p <probe> <user/program>`.

To better visualize the traces captured by NEUTRINO, we introduce a novel plot named Densified Memory Access Timeline (DMAT, §5), which improves the previous page reference map (strings) [11, 22] with physical time information and memory access density from parallelism. As illustrated in Fig. 1, DMAT expands new dimensions of observability compared with *hardware-dependent* profilers (Fig. 1B) and *kernel-exclusive* software profilers (Fig. 1C), enabling more comprehensive and intuitive GPU runtime analysis. For example, by comparing their DMAT profiles (Fig. 11), we can visually and quantitatively confirm that FlashAttn-v1 [20] improves memory efficiency and that FlashAttn-v2 [19] benefits from better pipelining.

We perform comprehensive evaluations (§6) to validate the trustworthiness, overhead, and applicability of NEUTRINO on profiling real GPU workloads. The results demonstrate that NEUTRINO ensures both execution correctness, *i.e.*, probing will not change the original execution flow, and profiling accuracy, *i.e.*, profiles are trustworthy. It also yields low overhead in both kernel slowdown (only 1.04x for most probes) and additional register usage (on average 4.11 more registers) Moreover, our extensive evaluations spotlight high system efficiency compared to other profilers and the capability to profile the whole model, even for LLMs. To showcase how NEUTRINO-profiled insights can help diagnose performance issues in GPU kernels, we conduct a case study (§7) on the impact of synchronization on GPU runtime behavior, which reveals an unnoticed tailing effect of shared GPU blocks on one compute unit and helps pinpoint different root causes of the performance bottlenecks.

NEUTRINO currently has limitations inherent to assembly-level probing, such as the inability to access unprogrammable hardware like caches. Nonetheless, as a fine-grained, versatile, and programmable framework for GPU kernel profiling, we envision NEUTRINO as a valuable tool for both research and industry communities. We have fully open-sourced NEUTRINO at https://github.com/open-neutrino/neutrino, and hope to foster a global community for its continuous development.

	Design		Fine-Granularity				Versatility			Programmability			
	Platform	Target	Grid	Block	Thread	Instructions	Value(Register)	Time(Clock)	Memory	Programmable	Persistence	Cooperativeness	Verification
Instrumentation-based GPU Kernel Profilers													
Neutrino	NVIDIA/AMD...	assembly	✓	✓	✓	✓	✓	✓	**DMAT**	DSL/TOML	eBPF-like Map	✓	✓
Nsight Compute [58]	NVIDIA	unclear	✓	✓	statistics	PC Sample	✓	✓	-	-	unknown	-	-
NvBit [93]	NVIDIA	machine code	✓	✓	✓	statistics	✓	-	PRM	C++	atomic	-	-
GTPin [80]	Intel	machine code	✓	✓	✓	✓	-	basic-block	PRM	C++	atomic	-	-
HIPAnlyzer [21]	AMD	compiler	✓	✓	✓	✓	-	basic-block	PRM	LLVM	event buffer	-	-
Hardware-based GPU System Profilers													
CUPTI [55]	NVIDIA	-	✓	✓	-	PC Sample	-	PC Sample	PM Sample	-	-	-	-
RGP/GPA [7]	AMD	-	✓	✓	-	PC Sample	-	PC Sample	PM Sample	-	-	-	-
CPU-side Software-based Profilers and Benchmarkers													
torch.profiler [87]	Independent	-	-	-	-	-	-	-	-	-	-	-	-

Table 1: **NEUTRINO vs. other GPU Profilers**. NEUTRINO is the only platform-independent runtime GPU kernel profiler and offers many unique features, *e.g.*, the DMAT, programmability via Python DSL or TOML, cooperative probes, eBPF-like maps, *etc*. PM stands for performance monitor, PC stands for program counter, and PRM stands for thread-local page reference map.

2 Background and Design Choice

GPU Profiling: Profiling builds roadmaps for performance engineering. Different from CPU profiling [12, 18] emphasizing sequential execution efficiency like branch prediction, GPU profiling prioritizes parallel execution scalability like compute unit utilization and throughput. Taking memory access as an example, CPU profiling cares about temporal locality like the working set [22], while GPU profiling focuses more on coalescing access among threads to utilize the bandwidth, opening unique challenges and research opportunities.

GPU Ecosystem: In modern computer systems, GPU has become a general-purpose computing unit (GPGPU) backed by a huge and complicated ecosystem covering numerous computing tasks, such as deep learning training [8, 27, 79] and inference [62, 94, 96]. However, from the perspective of compilation, the complicated ecosystem can roughly be divided into two branches: ❶ Ahead-of-Time (AOT) compiled operator libraries of hand-written codes [53] compiled by C++ compilers [6, 54], *e.g.*, ATen [8]; ❷ Just-in-Time (JIT) compiled domain-specific languages (DSLs) such as triton [91] compiled by LLVM [44] / MLIR [45]. As shown in Fig. 2, these two branches diverge only above the parallel assembly. Consequently, to build a general-purpose profiler, it has to be on or below the parallel assembly layer.

GPU Organization Hierarchy: As the *thread* and *memory* organization on GPU significantly varies from CPU for parallelism, we present some key differences between CPU and GPU below that are related to our design. First, parallelism on GPUs is hierarchical: 32 or 64[1] threads are grouped into a warp, the *scheduling unit* of GPU, *i.e.*, these threads share a single PC and must execute the same instruction at the same cycle. Warps are further grouped into blocks, the *concurrent execution unit*, *i.e.*, threads in the same block are executed in one physical compute unit (CU)[2] for communication and synchronization. Finally, blocks are grouped into grids mapped to the same GPU, which is the *management unit* for host side, and the measurement unit for kernel-exclusive profilers [87].

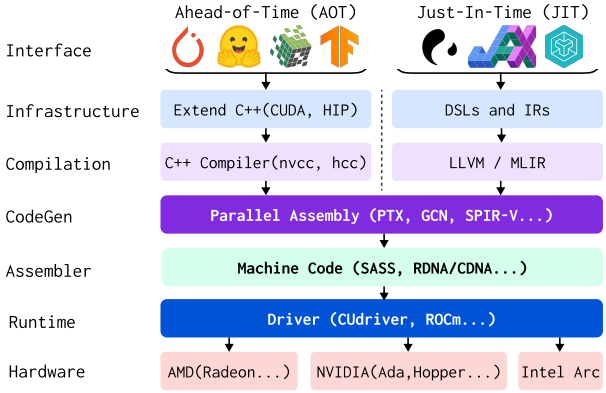


Figure 2: Complex GPU Ecosystem divided in two branches AOT (left) and JIT (right) unified at parallel assembly layer.

Similarly, the GPU memory is also hierarchically organized. First, each thread holds private registers (RMEM, at most 255 32-bit regs per thread on A100) as a primary resource. Blocks have CU-level shared memory (SMEM, at most 164KB on A100) as the temporary buffer for results and communications. Finally, there's GPU-level global memory (GMEM, 80GB on A100) for grid-level synchronization and kernel input/output.

GPU in the Operating System Paradigm: Positioned as an accelerator that communicates with the host OS via the driver, GPU programs are centralized with *kernel functions*, the entry for GPU computing whose content will be executed on the GPU while the rest of the program remains on the host CPU. Specifically, GPU kernel is considered *atomic* to the host OS, *i.e.*, the execution inside the kernel is managed by GPU hardware/firmware and is invisible and untouchable to the host OS, which prohibits observing GPU programs through mature OS technologies like ptrace or eBPF [24].

Apart from the host, profiling on GPU threads is also difficult as the systematic functionalities of GPU threads are

[1] NVIDIA GPUs group 32 threads into a warp, while AMD GPUs mostly group 64 threads into a wavefront (the different AMD's name of warp).

[2] Vendors may use different names for compute unit, *e.g.*, SM (Streaming Multiprocessor) for NVIDIA GPU or EU (Execution Unit) for Intel GPU.

highly limited. GPU programs are of direct execution without a management layer like the OS kernel. Particularly, there is no support for timer interrupts, a crucial feature for sampling-based profilers. Therefore, profiling techniques on traditional OSs, *e.g.*, sampling and scanning the stack frame, are not applicable to GPUs. Moreover, there are no commonly supported disk I/Os, making it troublesome to save results.

Assembly as a Probing Interface: The unique characteristics of GPU programming pose great challenges in building NEUTRINO, a fine-grained, versatile, and programmable GPU profiling tool that is *kernel-inclusive* and *hardware-independent*. The key question we need to answer is: at *which* layer should NEUTRINO be built and *how*? In this paper, among all the choices in Fig. 2, we choose the parallel assembly, such as PTX/GCNAsm [4, 60] designed to adapt to rapid changes in system and machine code, as the probing interface. Importantly, instead of static approaches, such as customizing profiling passes [21] via compilers or naive `asm()` in C, we employ a more capable but challenging approach, *attaching probe(s) at runtime*, which minimizes usage overhead (*e.g.*, recompiling) for changing or enabling/disabling probes. This design choice not only allows forward- and backward-compatibility, but also promises distinct advantages in various aspects:

❶ **Hardware-Oriented:** As a low-level interface, assemblies can capture hardware events that are important for performance analysis but hard to track with high-level languages. For example, there are only four related instructions in PTX for memory access, *i.e.*, `ld`, `st`, `cp.async`, and `tensormap`. In contrast, with objects and templates in CUDA C++, it is difficult to classify and capture all possible memory access.

❷ **Special Registers / Instructions:** Special registers of parallel assemblies contain useful runtime information for profiling. For instance, `hwreg` of GCNAsm tells which compute unit the thread is scheduled on, while PTX's special registers `%clock` (in CU-local clock cycles) and `%globaltimer` (in GPU-local nanoseconds) are helpful to measure the timestamp and can be used as instruction-level timers as shown in Fig. 3.

❸ **Compatibility:** As shown in Fig. 2, parallel assemblies are the *highest common layers* of the AOT and JIT compilations. For example, PTX is the common output of CUDA C++ compiled by gcc-based `nvcc` [54] and DSLs, *e.g.*, Triton backed by LLVM [86]. Thus, probing on assemblies can be compatible with most infrastructures, while compiler-based approaches are limited to the specific compiler or IR(s).

❹ **Coverage:** Compiler-based approaches require source code, assuming that users have located the poor-performing GPU kernel, which is uncommon as most programs have many kernels. Instead, runtime approaches can cover all user codes, capable of scanning poor-performing kernels.

The design choice of assembly-layer runtime probing also poses unique challenges, such as securing probes without compiler support, locating GPU code in the runtime, obtaining high-level contexts, *etc*. We overcome these challenges in NEUTRINO and enable it as a powerful GPU profiler.

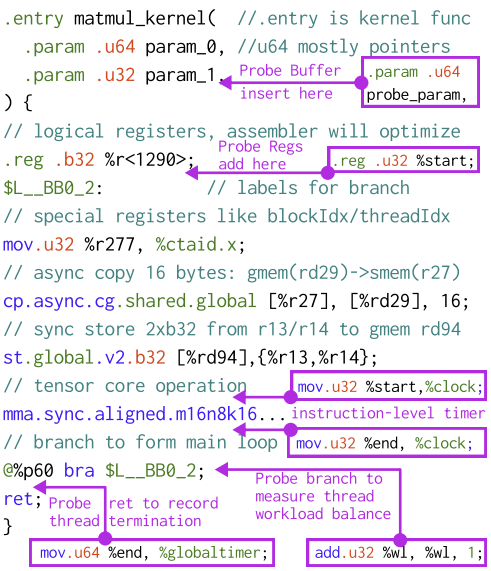


Figure 3: Parallel assembly example (PTX) with possible probing positions and corresponding functionalities.

3 NEUTRINO Design

NEUTRINO aims to formulate a simple yet powerful probe design like eBPF probe [24] to profile GPU kernels with the finest granularity to capture instructions, versatility to cover time- and value-profiling, and programmability for users to customize probe(s). Due to the massive parallelism of GPUs, NEUTRINO targets lightweight probes that only operate at tracepoints with the least disturbance to the remaining.

3.1 Programmable Probing Interface

As shown in Fig. 4, NEUTRINO features three key elements in its probe design: snippet, tracepoint, and structured map:

Snippet: Same as the probing target, NEUTRINO's snippet is assemblies, with some helpers such as `SAVE` for logging results and `OUT/IN1/IN2` for reading registers (instruction operands) for value profiling. Developers can also use other assembly features, especially `S_MEMTIME` for time profiling.

Tracepoint: NEUTRINO formulates probe tracepoints primarily at the finest instruction level, which assures both temporal accuracy and hardware granularity such as `wmma/mma` for tensor core operation. By grouping instructions, NEUTRINO's tracepoints can be extended to larger scales such as device function calls and thread start/end.

Map: Similar to eBPF [48], NEUTRINO's map explicitly structures the saving format to address the problem of persistence, a troublesome issue on GPU due to race conditions from parallelism and huge metadata from hierarchical organization. NEUTRINO mainly defines maps at two levels (§3.3): ❶ thread-level: *every thread* saves, for value profiling; ❷ warp-level: only *warp leader thread* saves, for time profiling.

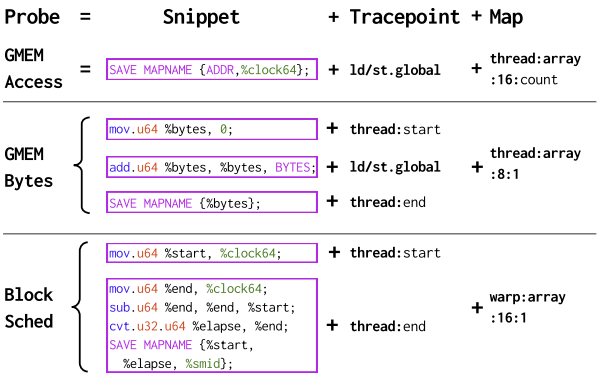


Figure 4: NEUTRINO programmable probing interface. Probes consist of snippet, tracepoint and structured map. Snippet can use helpers such as SAVE for storing values to NEUTRINO Map (§3.3). Multiple probes at different tracepoints can compose more comprehensive tasks like `block_sched`.

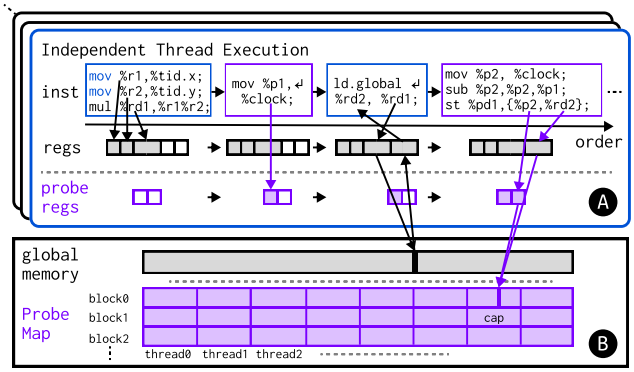


Figure 5: NEUTRINO probe execution model. ❶ Probes are executed virtually under time and resource (register and global memory) separation; ❷ NEUTRINO's Probe Map for race-free and metadata-efficient persistence: Each thread finds its segment of map via its `threadIdx` and `blockIdx`.

Beyond the three components for formulating customizable probes, the key design of NEUTRINO's programmability is *cooperativeness*: ❶ NEUTRINO probes of the same thread can cooperate by leveraging registers as temporal storage for advanced profiling tasks while maintaining efficiency as register usage is parallel in GPU cores. ❷ NEUTRINO probes can also cooperate with maps in GMEM, *i.e.*, different probes can contribute to and cooperate through the same map.

3.2 Virtualized Probe Execution Model

Since the GPU program is *static* to the OS, *i.e.*, all code (assembly) is loaded and known before execution, we choose to directly place probes in the original assembly without protection, *e.g.*, stack, to achieve cooperativeness. We identify that by doing so, NEUTRINO probes are still executed *virtually* from the original program. As shown in Fig. 5A, such virtualization is achieved via time and resource separation:

Time Separation: The time virtualization of NEUTRINO originates from the SIMT execution model of GPGPU, where parallelism happens among threads while execution within a thread is generally sequential with one instruction at a cycle. Thus, as probes are directly inserted into assemblies, their time separation from the original program will be guaranteed.

Resource Separation: Similar to CPU, GPU threads also have *thread-private* registers as their primary resources, which contain intermediate results from ALU, addresses of shared or global memories, *etc*. NEUTRINO virtualizes the probe registers by separating an independent register group, as well as other resources like GMEM. Thus, NEUTRINO probes can avoid affecting the original program's resources, and the execution flow. It is worth noting that NEUTRINO's probe register group is declared logically at the assembly level rather than physically. Therefore, NEUTRINO may not necessarily introduce extra physical register usage (Tab. 3) because the declared logical registers will be integrated into physical registers by the assembler in register allocation, with independence between probes and the original program preserved by dependency tracking algorithms [39, 72].

3.3 Structured Map for Persistence

Persistence is a critical challenge for GPU profiling. Although thread executions are parallel and independent, the underlying memory system is shared, leading to race conditions among concurrent savings. Thus, previous solutions [80, 93] widely use *atomics* for separating persistence spaces, which can become inefficient under massive parallelism. Moreover, the GPU hierarchical organization creates rich *metadata* such as `threadIdx` and `blockIdx` (24 bytes), which is informative for analysis but can be storage-hungry [93].

Inspired by the design of lock-free per-cpu eBPF maps [48] and event buffers of HIPAnalyzer [21], we explicitly structure NEUTRINO map to an `ndarray` layout, as shown in Fig. 5B, with the shape determined by launch configurations (`blockDim`, `gridDim`) and the map definitions (`level`, `type`, `size`, `cap`). This enables race-free saving as each thread has an independent segment and reduces storage pressure as most metadata can be inferred rather than directly saved. NEUTRINO mainly formulates maps at two levels:

Thread-Level: The thread-level map is mainly designed for *value profiling* where every thread independently saves data. Its layout is in the form of `[#Grid, #Block, cap]` and each element is of `size`. `#Grid` and `#Block` can be inferred by launch config `gridDim` and `blockDim`, respectively. `cap` specifies the maximum number of savings per thread, which can be set to a suitable value, or measured dynamically by running a counter probe[3] in the runtime.

[3] A probe of `count+=1` attached at tracepoints, and after execution, the value of `count` can be regarded as `cap` (only applies to *pure* kernel functions).

Warp-Level: The warp-level map simplifies the thread-level map for time profiling with layout `[#Grid, #Warp, cap]`. Because threads within the same warp are scheduled together for the same instruction, recording event timestamp only needs one thread inside the warp other than all, which can significantly reduce the memory and storage pressure.

Based on these two levels, NEUTRINO can extend different types of maps, such as the simplest `array`, or advanced `ring` (ring buffer) and `hash` to support versatile user needs.

3.4 Verification for Security

Verification [28, 84] has been proven vital for programmable probes [24] as unsafe probes can break the execution flow of the original program and invalidate the profiling. The verifier can also help guide developers in writing correct probes. In NEUTRINO, we identify and prevent three key security issues:

Overwrite Original Registers: As discussed in §3.2, GPU threads use registers as primary resources. Thus, modifications of registers used by the original program are unsafe. For example, modifying a register holding address to global memory could lead to illegal memory access (Fig. 6A). Thus, NEUTRINO requires probes to use independent register groups, and prohibit probes that modify original registers.

Program Misorder: Although the SIMT program model guarantees that instructions within the thread are executed linearly, there are flow control instructions, such as `S_BRANCH` (GCNAsm) or `bra` (PTX), that may change the execution order, which are unsafe for probes as they may break the original execution order (Fig. 6B). Thus, NEUTRINO prohibits probes from instructions that change the execution flow.

Shared Memory: As an important factor for acceleration, the shared memory has been highly optimized for storage [17, 91] and access efficiency [89]. Thus, additional shared memory usage from the probes may greatly affect or fail the execution if the original usage is already at the hardware limit. Thus, NEUTRINO prohibits probes from using shared memory.

4 NEUTRINO Implementation

We implement NEUTRINO in Linux for NVIDIA GPU with CUDA driver and AMD GPU with ROCm/HIP runtime. Our implementation consists of three major components: ❶ a hook driver (§4.1, ≈2,500 lines of C code) to provide runtime support for assembly tracking, code caching, *etc*; ❷ a probe engine (§4.2, ≈2,000 lines of Python code) to instrument parallel assemblies; ❸ a DSL compiler (§4.3, ≈1,000 lines of Python code) to translate probes in platform-agnostic Python Tracing DSL into platform-specific assemblies (PTX for CUDA and GCNAsm for ROCm/HIP). We implement the core probing engine and the DSL compiler in Python to make the infrastructure more approachable and extensible for developers. Besides, we also provide utilities (§4.4), such as ecosystem integrations, analysis code generations, *etc*.

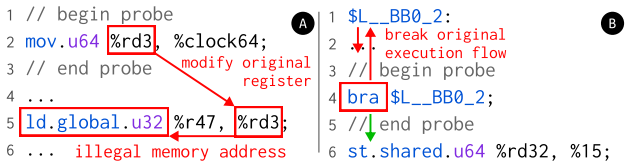


Figure 6: NEUTRINO verification against two unsafe operations: ❶ overwrite original regs; ❷ change execution flow.

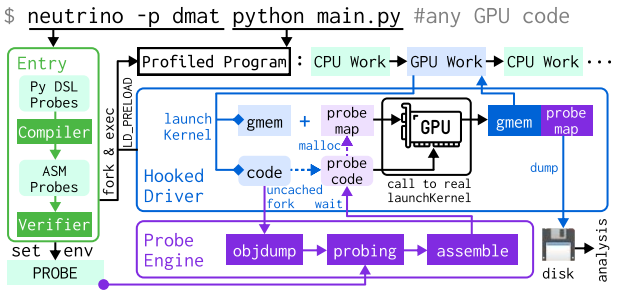


Figure 7: NEUTRINO workflow. Entry loads and JIT compiles (§4.3) probes, and injects hook driver. Hook driver (§4.1) captures GPU workloads, invokes probe engine (§4.2), allocates probe buffers, launches probed kernel, and dumps results.

Finally, we wrap these modules as a command-line interface similar to `bpftrace` [13] and `valgrind` [74]. As shown in Fig. 7, when a user invokes `neutrino` with the probe `dmat` via `-p/--probe` argument, the entry will load, compile, and verify the probe (`.py`) into platform-specific assembly (`.asm`) wrapped in TOML [68]. Then the entry will set environment variables, such as `NEUTRINO_PROBE` for probe contents and the special `LD_PRELOAD` to inject the hook driver, and `fork` a child process to launch the workload. Throughout the execution, the hook driver will continuously capture the GPU workload, particularly the GPU kernels launched. For each uncached GPU kernel, the hook driver will invoke the probe engine, which objdump, probe, and reassemble the kernel. After loading back the probed kernel, the hook driver allocates probe buffers on CPU and GPU, and launches the probed kernel. Upon completion, the hook driver will dump probe buffers containing metrics and give back control to user programs.

4.1 Hook Driver

Though drivers in OS are mostly referred to kernel extensions exposed via `read/write/ioctl` syscall, such as `nvidia.ko`, most vendors also maintain higher-level user-space drivers as shared libraries such as `libcuda.so` or `libamdhip.so` in Linux. These driver shared libraries are often highly complex and closed-source. However, given that symbols in ELF are resolved via their signatures, we can build a clever hook driver by defining all functions with matching signatures and using `dlfcn` internally to locate and call the real function from the actual driver (More details in Appendix B). Compared with

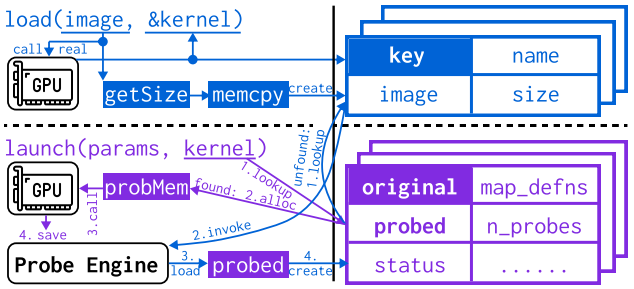


Figure 8: NEUTRINO hook driver infrastructure. Hook driver catches `load` and `launch` API for loading images and launching kernels on GPU, respectively. Hook driver maintains two hash-based [33] storage for image (upper) to map kernel to binary image, and kernel (lower) to avoid repeated probing.

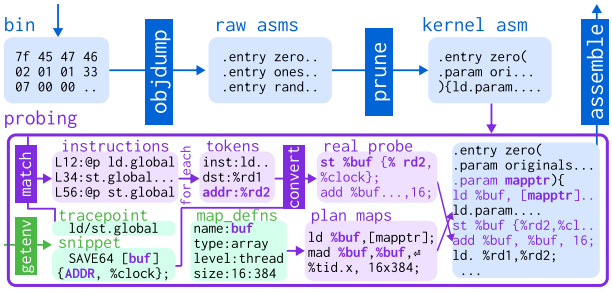


Figure 9: NEUTRINO probe engine workflow. Probe engine objdumps and prunes the binary, and finally reassembles it. In probing, it uses *map defns* to plan probe buffers and *tracepoints* to match instructions. It also fills helpers in *snippets* with tokens before injecting them into assemblies.

other approaches like eBPF `uprobe` [24], our hook driver is safer and more flexible as all code is executed in the user mode, supporting `fork/wait` that are important for interactions with the probe engine. We leverage the hook driver to provide the following supports:

Code Tracking: Compared with CPU code implicitly loaded by the OS, GPU code, in the ELF [25] or FatBinary [56] format, requires an explicit `load` via `cuModuleLoad`. Other functions, such as `cuModuleGetFunction` may also be applied to locate the specific kernel from the module. We hook these APIs (Fig. 8) to capture all images loaded, kernels extracted, and the mapping from kernels to code. Each image is `memcpy`ed into the *image storage* with size from its header to avoid being `free`d by resource management of user programs.

Runtime Probing: As the execution is non-local, launching GPU kernel functions is not simply adding stack frames but requiring an explicit driver call to `cuLaunchKernel` or `hipModuleLaunchKernel`. We hook these APIs to provide runtime support: ❶ It searches for the probed kernel in the *kernel storage* via the original kernel (a pointer); ❷ It `allocates` probe buffer(s) according to the metadata; ❸ It `launches` the probed kernel and `synchronizes` for the finish, *i.e.*, probe buffers on GPU are of metric readings; ❹ It `memcpy`s probe buffer back to CPU, then `fwrites` and `frees` probe buffers.

More importantly, when the probed kernel is not found in the *kernel storage*, the hook driver is also responsible for interacting with the probe engine: ❶ It searches for the binary containing the kernel in the *image storage*. ❷ It `fwrites` the binary in the directory and `forks` a subprocess to invoke the probe engine. ❸ After `waiting`, it looks up the directory and `loads` back the probed kernel and metadata, e.g., number of probes. ❹ Finally, the probed kernel and metadata will be added to *kernel storage*. Failed kernels will also be added to *kernel storage* with `status=false` to avoid repeating.

Other functions are unhooked and we auto-generate them by parsing the header `cuda.h/hip_runtime.h` and the symbol table of the shared library, such as `libcuda.so`.

4.2 Probe Engine

As shown in Fig. 9, NEUTRINO probe engine first `objdumps` the dumped GPU binary to extract parallel assemblies in text format. Then it will use the kernel name to match and prune the *many-kernel* raw assemblies into a *single-kernel* assembly while keeping global definitions and device functions.

Next, NEUTRINO will process and add the probes read from environmental variables, involving the following steps: ❶ It plans probe map(s) that directs each warp/thread to its segment(s) of map(s) according to the *map definition* (§3.3, `level/type/size/cap`) and thread indexes, *e.g.*, `threadIdx`, as detailed in Appendix C. ❷ It coarsely parses the kernel assembly into parameters, register declarations, and instructions. Then it matches the *tracepoints* to specific instruction(s). ❸ It thoroughly parses each matched line (e.g., `ld.global.u64 %rd1, [%rd2];//%rd1=*%rd2`) into tokens, such as opcode (`ld.global.u64`) and operands `%rd1, %rd2`. Then it fills the helpers in *snippets*, such as `ADDR`, to the real register `%rd2`. Finally, it places snippets before/after matched instructions, as well as the map addresses at the end of kernel parameters, and the assembly of map planning at the kernel beginning.

After probing, it converts the probed assemblies into machine code via assemblers such as `ptxas` [61]. The probe engine will also save kernel metadata useful for the hook driver, such as the probes, maps, callbacks, *etc*.

4.3 Probe DSL and Compiler

A practical issue of the probe engine is that probes are of assemblies, a low-level and hardware-dependent language. Direct assembly programming may be less friendly for general developers. Thus, to enhance NEUTRINO's hardware independence and usability, we propose a minimalistic Python domain-specific language (DSL) as the high-level interface for NEUTRINO probes, similar to bpftrace [13] for eBPF [24]. Note that DSL is optional for NEUTRINO, experienced developers can still handcraft assemblies for advanced usages.

As shown in Fig. 10, the NEUTRINO DSL closely follows the Python syntax, allowing users to declare probes with the `@probe` decorator over functions with tracepoints specified as decorator arguments and the snippet as the function body. Similarly, maps can be declared with `@Map` decorator with the structure defined as class members. Moreover, contexted probe registers shared across probes can be defined via value assignment syntax with types annotated in the global scope. NEUTRINO probes are not allowed to use other functions like `open`. Instead, we provide helper operands like `nl.addr` for reading registers and helper functions like `nl.clock()` and `Map.save()` for getting device-side clocks and saving results.

This DSL will be *just-in-time* compiled into the platform-specific assembly-based probes in two steps, with a compilation example of Fig. 10 in Appendix D: ❶ It uses Python's `ast` module to parse and transform the Tracing DSL to an intermediate representation (IR) similar to the eBPF ISA [90]. ❷ The IR will be translated into platform-specific assemblies, *i.e.*, PTX Assembly and GCNAsm, and helper operands will be preserved for the probe engine. We design the IR to be eBPF-like, potentially enabling reuse of mature eBPF toolchains, such as the reputable eBPF verifier [28].

```python
from neutrino import probe, Map
import neutrino.language as nl
CALLBACK = "block_sched.py" # for trace analysis
# declare maps for persistence
@Map(level="warp", type="array", size=16, cap=1)
class block_sched:
    start:   nl.u64
    elapsed: nl.u32
    cuid:    nl.u32
# declare probe registers shared across probes
start:   nl.u64 = 0 # starting clock
elapsed: nl.u64 = 0 # elapsed time, initialized to 0
# define probes with decorator
@probe(pos="kernel", level="warp", before=True)
def thread_start():
    start = nl.clock()
@probe(pos="kernel", level="warp")
def thread_end():
    elapsed = nl.clock() - start
    block_sched.save(start, elapsed, nl.cuid())
```

Figure 10: NEUTRINO DSL probe example. Each *warp* has probe register, *e.g.*, `start`, and saves a `block_sched`.

4.4 Utilities

Ecosystem Integration: Solely hooking onto the driver or probing at the assembly would lack high-level info like tensor shapes that are useful for analysis. Thus, we also implement utilities for ecosystem integrations like PyTorch [8] by Python `sys.settrace` to expose high-level tensor information.

Benchmarking Mode: To evaluate system overhead and provide time alignment for heavy probes, NEUTRINO provides a benchmark mode that launches both the probed kernel and the pruned kernel (stripped of probes and assembled with identical configurations) with CUDA/HIP event timers to benchmark the additional execution latency caused by probes.

Analysis Codegen and Callback: To facilitate analysis, NEUTRINO supports generating tracing parsing code in Python based on the map definition (such as Fig. 10) to facilitate users extracting information from traces, with an example in Appendix E. Moreover, NEUTRINO supports adding callbacks (such as Line 3 of Fig. 10) for automated posterior analysis.

Source Code Annotation: To give more precise control, we implement a source code annotation tool in NVTX-like API [59]. The probe engine can look up the lineinfo (special comments, e.g., `".file 1 example.py"` and `".loc 1 33 45"`) and cross-reference with the source code (line 33 of 1st file, *example.py*) to include or exclude the instruction.

4.5 Extending NEUTRINO to Other Platforms

Though the current implementation only supports NVIDIA and AMD GPUs, NEUTRINO can be extended to other platforms, such as Intel oneAPI [26]. NEUTRINO's hardware independence originates from its design on the two common components across platforms: parallel assembly to accommodate rapid architecture-level evolution, and the driver to control the execution from the host OS.

In practice, to extend NEUTRINO to other platforms, one needs to implement the hook driver, probe engine, and *(optional)* DSL compiler backend. For the hook driver, as most functionalities are standardized to platform-agnostic modules, we expect most changes to be around API renaming and debugging, *e.g.*, `cuLoadModule` to `hipLoadModule` for ROCm/HIP support. Regarding probe engine, a new parser and matcher for the different assembly syntax (*e.g.*, GCNAsm [4]) shall be needed, but the overall infrastructure (Fig. 9) remains unchanged. The DSL compiler needs a backend to translate our eBPF-like IR into assembly. We expect this would be similar to how Triton [91] supports new hardware with extended codegen (like our probe engine and DSL compiler) and launcher (like our hook driver).

4.6 Usage: Putting It All Together

The above components shape a user-friendly and easy-to-use profiling tool of NEUTRINO. Compatible with many frameworks such as PyTorch [8], Triton [91] and JAX [27], the usage of NEUTRINO is as simple as `bpftrace`, with many built-in tools such as `block_sched` (Fig. 10) to check the block scheduling cost of kernels. NEUTRINO's user-friendliness is best demonstrated through a simple example, where we try to profile the following line of PyTorch code and gain insights:

```
torch.zeros((4096,4096),torch.float16,device="cuda")
```

To do so, a user just needs to run the NEUTRINO CLI with the `-probe/-p` option:

```
$ neutrino -p block_sched python -c "torch.zeros(...
```

Then, when finished, traces will be placed in a directory with a print-out message from the analysis callback as follows:

```
vectorized_elementwise: # kernel name, truncated
No.block:32768 Exec:680869 Sched:142674 (cycle/SM)
```

Here the `vectorized_elementwise` kernel [88], widely used by unary tensor operations, is used for initializing allocated memory with zeros. We measure the scheduling time by simulating the block dispatching to CUs. For every recorded block on the CU, if its `start` clock is greater than any existing block's `end` clock (`start + elapsed`), then a block replacement happens with scheduling cycles estimated by *the next* `start` - *the previous* `end` and execution cycles measured by the previous `elapsed` (Complete code in Appendix E).

The profiling results reveal a surprising $\sim$ 20% time spent on scheduling blocks to physical SMs as the kernel launches a huge number (32,768) of blocks and the execution time of each block is relatively small. Based on the insights, one can optimize the performance by reducing the number of blocks launched by the kernel. To do so, we can use the CUDA `memset` or instruct LLMs to write a persistent kernel (examples in Appendix F) that fixes the number of blocks to the hardware limit. To apply customized initialization, we replace `torch.zeros` with `torch.empty` that allocates memory without initialization, and add a line of `cuMemset` or `zero_persistent`. This modification offers $\sim$ 28% speedup:

```
# Original kernel time: 34,493 ns
torch.zeros((4096,4096),torch.float16,device="cuda")
# Updated memset time: 24,630 ns
t=torch.empty((4096,4096),torch.float16,device="cuda")
driver.cuMemsetD16(t.data_ptr(), 0, 4096*4096)
# Updated zero_persistent kernel: 24,891 ns
zero_persistent(t) # code in Appendix F
```

The above one-line debugging example demonstrates the capability and simplicity of NEUTRINO to pinpoint performance bottlenecks within the kernel. Note that NEUTRINO also supports more advanced usage beyond the above example, including profiling the whole model with other interesting tools:

```
$ neutrino -p tensorop_count # number of tensor op
$ neutrino -p gmem_bytes # number of GEMM bytes used
$ neutrino -p dmat # draw DMAT Plot
$ neutrino -p <to/be/contributed/by/you>
```

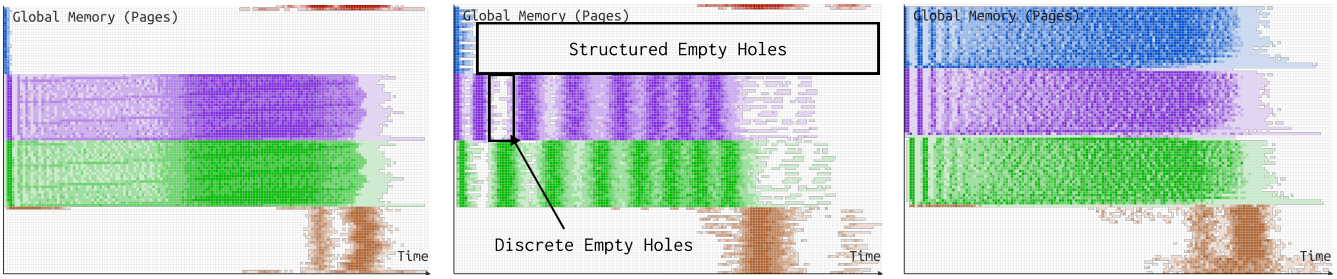
5 NEUTRINO Visualization

In this section, we introduce the Densified Memory Access Timeline (DMAT) plot, an insightful visualization of GPU runtime workload behavior, as presented in Fig. 1 and Fig. 11. DMAT plot is inspired by the *page reference map* [11, 22], which shows the virtual time as the x-axis and page accesses as the y-axis, where a point represents an access to the page at the time. As a reference standard, page reference maps have proven to be a useful tool for research on virtual memory management [11] and replacement algorithms [10].

To accommodate the massive parallelism of GPU, DMAT extends the original page reference map in two perspectives:
Physical Time: Page reference maps [11, 22] and general memory tracers [66, 74] commonly use a *thread-local* auto-incremental index as the virtual time denoting the access order. However, we find that the virtual time is insufficient under parallelism where each thread only holds part of the entire page reference map. Because starting times and execution paces of threads diverge, there will be unavoidable misalignment among the virtual time when aggregating page reference maps from individual threads to form the complete trace.

Hence, for DMAT we use the *device-side* physical time to provide reliable aggregation. Specifically, we provide two types of DMAT: ❶ Normalized to the starting clock (Fig. 1, Fig. 11) of unsynchronized CU-level clocks, suitable for analyzing algorithm behavior; ❷ Synchronized to a less accurate (MHz) GPU-local timer (Fig. 15), representing actual memory access for hardware/cache analysis (Appendix G).
Page Access Density: Previous page reference maps are in 2D, where a point denotes an access at the time to the page. However, for highly parallelized environments, there are likely many concurrent accesses to the same page at the same time from multiple threads. We record such parallel access intensity as *density* and mark it as color depth to distinguish it from the temporal frequency in traditional page reference maps [11, 22], highlighting the new informative dimension spanned for analyzing the effect of parallelism.

The proposed DMAT not only facilitates traditional memory analysis on GPU (*e.g.*, data races, access anomalies), but also features unique benefits for GPU runtime analysis:
■ **Color Depth:** The color depth in DMAT denotes the density of parallelization. When aligned to the GPU-local timer, it reflects the real memory load, with shallow regions pinpointing uncoalesced access among threads and excessive intensity indicating potential memory I/O contention. When aligned with the starting clock, color depth reflects the divergence among threads where pale patterns usually indicate divergent thread executions or imbalanced workloads that may waste the computing power.
■ **Empty Holes:** Empty holes in DMAT demonstrate that pages remain unused for a significant amount of time, which includes two cases (Fig. 11b): ❶ Discrete empty holes usually reflect computing within CUs. The duration reflects the operational intensity [95] per main loop, where a too long empty holes might be inefficient in pipelining [20]; ❷ Structured empty holes usually reflect algorithm improvements, while extra-large structural empty holes may reflect time fragmentation and optimization opportunities.



(a) Flash-Attn v2 (Triton) [19] with shared block (b) Flash-Attention-v1 [20] (c) Memory-Efficient-Attention [69]

Figure 11: NEUTRINO DMAT plot (captured on RTX3080, Appendix A) for different attention algorithms, which exhibit distinct memory access patterns. (a) differs from Fig. 1 with exclusive SM. By comparing the DMAT of different algorithms, we can visually identify the improvement of (b) FlashAttn-v1 [20] *w.r.t.* (c) Memory Efficient Attention [69] comes from I/O efficiency, while the gain of (a) FlashAttn-v2 (Fig. 1) comes from better pipelining, both consistent with their respective claims.

6 NEUTRINO Evaluation

We conduct evaluations to answer the following questions on the reliability and usability of NEUTRINO: Are NEUTRINO's results trustworthy (§6.1)? How much performance and resource overhead does NEUTRINO introduce (§6.2)? How effective is NEUTRINO in profiling real applications (§6.3)?

Our evaluation is performed on two platforms, NVIDIA A100 80GB GPU and NVIDIA RTX4090 24GB GPU. NEUTRINO is compiled with gcc-11.4.0. Other software used includes Python 3.11.4, CUDA 12.6, PyTorch 2.5.0, Triton 3.1.0, CUTLASS 3.5.0, and Ubuntu 22.04.

6.1 Correctness Validation

We identify and verify two types of correctness: ❶ **Execution Correctness** that ensures the probing will not alter the original execution flow; ❷ **Profiling Accuracy** that ensures the metric reading from NEUTRINO is reasonable and correct.

Execution Correctness: We validate the execution correctness by comparing output differences between the probed and original kernels, as any difference in execution flow will likely change the output or corrupt the system. To do so, we run each test twice, one with NEUTRINO probes and one without probes (*i.e.*, the original execution) under identical input and configurations. The results show no significant differences between the probed and original outputs.

Profiling Accuracy: We verify the profiling accuracy from two perspectives. For metrics overlapped with other profilers like Nsight Compute [58], (`block_sched`, `gmem_bytes`, and `tensorop_count`), we run the same program separately with NEUTRINO and Nsight Compute, and compare their metric readings. The captured results give consistent metric readings.

Regarding new metrics beyond the scope of existing profilers, particularly DMAT (§5), we validate their correctness by profiling carefully designed "micro-benchmarking" kernels having expected theoretical metric values, and comparing with the actual metric readings. We design micro-benchmarking kernels using `ld/st` instructions with L1 cache disabled, for memory access, and spin-based sleeping to controllably simulate computation inside CUs. We implement several memory access patterns: Linear, Strided, Gather, Scatter, and Random. We emulate these access patterns on the CPU, with addresses, normalized to base addresses, derived from the access pattern's code, and intervals (in cycles) between consecutive accesses estimated by the sleeping time and memory latency.

We compare theoretical estimations and actual readings to evaluate: ❶ Address Consistency: Calculated as Hamming distance between emulated and profiled address sequences; ❷ Clock Errors: Measured as the differences of intervals between consecutive accesses within threads; ❸ DMAT Similarity: Computed as RMSE between DMAT (treated as matrices). Tab. 2 demonstrates that DMAT can correctly capture memory addresses (with Hamming distances of zeros) and achieve *less than 200* cycles time resolution (< 7% of the loop time). Accumulated clock errors will lead to a misaligned timeline for DMAT, resulting in high RMSE errors, which are marginal for coalesced accesses (*e.g.*, `stride`), yet become considerable for uncoalesced accesses, *e.g.*, $\sim$ 60% error for `linear`. It is because uncoalesced accesses have a lower average intensity ($\sim$ 16) as normalization bases and experience more variable memory latencies ($\sim$ 190 cycles clock errors). As detailed in Appendix H, large DMAT errors are mainly due to the static emulation not considering memory system dynamics, *e.g.*, cache, rather than profiling, which we reserve for future work.

Table 2: DMAT micro-benchmark *w.r.t.* theoretical metrics.

Kernel	Address Consistency Hamming Distance	Clock Errors Mean (Normalized)	DMAT Similarity RMSE (Normalized)
`linear`	0	190.1 (6.30%)	9.54 (59.62%)
`stride`	0	96.20 (3.42%)	277.4 (5.21%)
`gather`	0	58.11 (2.70%)	33.80 (0.44%)
`broadcast`	0	65.19 (3.01%)	192.5 (2.64%)
`random`	0	179.8 (5.98%)	221.7 (4.86%)

Table 3: Kernel slowdown and additional physical register usage of NEUTRINO: Kernel slowdown is normalized to original kernel latency and additional register usages are averaged based on assembler [61] debug information. NEUTRINO might lead to kernel speedup on lightweight probes, *e.g.,* 0.9868x speedup of `gmem_bytes` on `GEMM`, and we discuss this abnormal effect in §8 and Appendix K. `dmat` probe leads to different degrees of slowdown on different kernels, as discussed in depth in Appendix I.

		`block_sched` `warp:array:16:1`		`gmem_bytes` `thread:array:8:1`		`tensorop_count` `thread:array:8:1`		`mem_trace` `thread:array:16:count`	
		Kernel Slowdown	Additional Registers	Kernel Slowdown	Additional Registers	Kernel Slowdown	Additional Registers	Kernel Slowdown	Additional Registers
CUTLASS	Standard GEMM	0.9997x	+4	0.9868x	+4	0.9873x	+4	8.8933x	+1
	Stream-K GEMM	1.0050x	+12	1.0022x	+12	1.0034x	+12	10.3709x	+6
	Conv2D	1.0327x	+0	1.0934x	+0	1.1061x	+0	2.7463x	+28 (spill)
Triton	Group-GEMM	0.9804x	+8 (spill)	0.9798x	+8 (spill)	0.9796x	+8 (spill)	8.3589x	+516 (spill)
	Flash-Attnv2	0.9999x	+2	1.0257x	+2	1.0256x	+2	2.9392x	+1.50
PyTorch	Batch/LayerNorm	1.0251x	+4.58	1.0951x	+3.54	1.0981x	+3.25	5.4392x	+4.67
	SoftMax	1.0004x	+4.5	1.0006x	+3.5	1.0006x	+3.5	13.1630x	+8
	Sum	1.0653x	+4	1.0389x	+2	1.0390x	+2.67	4.4597x	+8.67
	Max/AvgPool	1.0406x	+3	1.3882x	+2	1.3875x	+2	7.2559x	+4
	Embedding	1.0068x	+6	1.0009x	+4	0.9995x	+2	9.1155x	+6
	Gather	0.9596x	+5	0.9957x	+2	0.9782x	+2	5.5659x	+6

6.2 Profiling Overhead

We evaluate two types of profiling overhead: ❶ **Performance Overhead**: the kernel slowdown due to probes can compromise the accuracy of time-related profiling; ❷ **Resource Overhead**: additional registers from probes, which may affect block dispatching or even lead to register spilling.

Performance Overhead: We formulate the kernel overhead as the slowdown from probe instructions normalized to the execution time of the original kernels. For better accuracy, we evaluate the kernel execution time via device event timers (§4.4). Results presented in each left column of Tab. 3 demonstrate the efficiency of NEUTRINO with controllable latency (1.04x on average) on lightweight probes *i.e.*, `block_sched`, `gmem_bytes` and `tensorop_count`, while heavy probes such as `dmat` introduce considerable slowdown (7.12x on average). And our analysis (Appendix I) shows that DMAT's frequent memory I/O is the cause of the slowdown, and degrees of slowdown depend on the percentage of memory accesses and kernel execution time. Moreover, we identify that lightweight probes could *abnormally* accelerate the program, and our analysis (Appendix K) finds that it is because probe instructions can lead assemblers to better instruction flow (+5.88% IPC) and thus better performance (up to 0.94x faster).

Resource Overhead: We formulate the resource overhead as the difference in the number of *physical* registers used by the probed kernels compared to the original kernels. Results shown in each right column of Tab. 3 demonstrates the low overhead for NEUTRINO probes with 3.78 more registers used in lightweight probes and 5.09 more registers used in the heavy `dmat` probe. The phenomenon that each probe defines the same number of *logical* registers but uses varying and fewer *physical* registers also confirms the effectiveness of our design of using logical registers rather than physical ones, allowing potential optimization by the assembler.

6.3 Extensive Study

We further conduct two extensive studies evaluating NEUTRINO's applicability in evaluating real-world workloads:

GMEM Usage in Model Profiling: In practice, developers need to profile the whole model, rather than a single kernel, to locate potential performance issues. GMEM usage becomes a constraint here as most GMEM is occupied by model parameters. Thus, we conduct an intensive test on NEUTRINO's maximum GMEM usage in end-to-end profiling on the whole model inference pass. We select ResNet [34], Stable-Diffusion [71], Mamba-1.7B [31], and Llama3-1/3/8B [29].

The result shown in Fig. 12 demonstrates the efficiency of NEUTRINO's memory usage, with lightweight probes being at least an order of magnitude smaller of the original memory footprint, especially for transformers like Llama. GMEM usage of the heavy `dmat` is mostly remains within the original memory usage, even under a large batch size (256). Moreover, by comparing the results of Llama-1B/3B/8B, we observe that the proportion of NEUTRINO's GMEM usage relative to the original usage decreases surprisingly as the model scales. This finding demonstrates that the growth of NEUTRINO's memory requirements is slower than model size scaling and highlights the usability of NEUTRINO in profiling larger models.

Profiler Exposed Latency: In addition to *kernel slowdown*, profilers also expose other noticeable latencies, including *Prologue* for allocating probe maps and calling the probe engine and *Epilogue* for copying back probe maps and saving traces to disk. The sum of these is the latency exposed to the upper layer. To evaluate the overall profiling efficiency of NEUTRINO, we compare their exposed latency with Nsight Compute [58] on overlapped metrics via application benchmarkers [8, 89, 91]. Results presented in Fig. 13 highlight the reduction of exposed latency and the efficiency of NEUTRINO's system design and implementation.

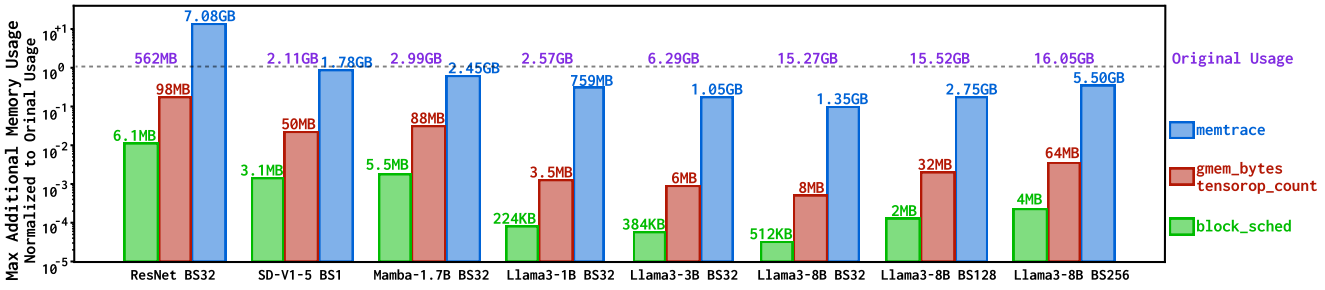


Figure 12: Max probe memory usage of NEUTRINO in profiling model forward. Probe memory usage is log-scaled to original memory usage ((labeled in purple). `gmem_bytes` and `tensorop_count` have the same map definition and the same usage.

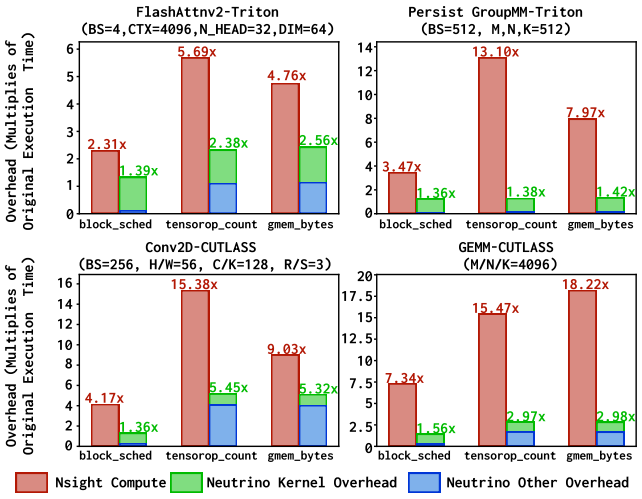


Figure 13: Exposed latency comparison: NEUTRINO and Nsight Compute [58]. NEUTRINO latency is decomposed into prologue (<1%), kernel and epilogue latency.

7 Case Study with NEUTRINO Insights

We envision NEUTRINO to be a useful tool for GPU performance engineering, paving the way for optimizing ML systems with fine-grained runtime insights. This paper mainly focuses on the design and implementation of NEUTRINO, and thus, how to leverage profiles is not our primary goal. Nevertheless, to showcase how one could exploit NEUTRINO and DMAT plots for previously unavailable insights, we carry out one case study on the impact of synchronization:

Similar to hyper-threading on CPU, GPU SM Sub-Partition (the execution unit) also maintains multiple candidate warps (the scheduling unit in GPU), and in each cycle, one warp will be selected by the warp scheduler to run. Such a design can reduce blocked waiting of instruction completion, particularly for non-local operations like memory I/O, as other candidates can be scheduled to utilize the core. In practice, there could be two cases for these warps: ❶ All warps belong to the same block that might need synchronization; ❷ Warps belong to different blocks that are mutually independent.

To identify the potential runtime difference with respect to the difference in synchronization, we create a controlled pair on Flash-Attn-v2 [19] implemented by Triton [91]. We leverage Triton auto-tuner to create two kernels: ❶ Exclusive blocks: 128x128 tile, 2 stages, 8 warps, 1 block per SM (2 warps on each SMSP are of the same block); ❷ Shared blocks: 128x64 tile, 2 stages, 4 warps, 2 blocks per SM (2 warps on each SMSP *can be* of different block). The DMAT plot (aligned with kernel start time) traced by NEUTRINO on A100 are presented in Fig. 1A and Fig. 11a, respectively. Although these two configurations impose the same computational load on the SM and offer similar throughput, we can find that their memory access patterns are significantly different.

For exclusive blocks (Fig. 1A) with frequent synchronization, the memory access pattern is structured with a regular access pattern to `K` and `V`, confirming that the algorithm [19] uses one load for both `K` and `V` per main loop. However, for shared blocks with independent synchronization (Fig. 11a), the memory access pattern is unexpectedly unstructured with many tailing blocks (light-colored parts on the right).

To validate the tailing effect, we conduct more in-depth analysis via the `block_sched` probe (Fig. 10) to analyze the elapsed time of blocks. From the CDF of elapsed time in Fig. 14A, we identify that the elapsed time of exclusive blocks is highly consistent, but varies significantly for sharing blocks (Fig. 14B) with a tailing effect of up to 24.69%[4].

To further explore the tailing latency, we take a closer look at the program execution progress via probing the `bra` instruction (Fig. 3) that redirects the program to form the *main loop*. This probe samples program execution via recording the timestamp at each `branching`, which can be used to recover the timeline of the block's working progress as shown in Fig. 14D. Furthermore, by taking the difference between sampled timestamps, we can further recover the intra-block-level throughput (TFLOP/s) timeline (presented in Fig. 14C).

From the sampled progress and throughput timeline, we observe an interesting phenomenon that every shared block

[4]The differences in completion cycles in Fig. 14A (∼220000) and Fig. 14B (∼430000) is from the FlashAttn-v2 algorithm that N is the sequential dimension, so the latency of two 128x128 tiles (Fig. 1) is equivalent to that of two 128x64 tiles (Fig. 11a), both handling 256 feature dimensions (M).

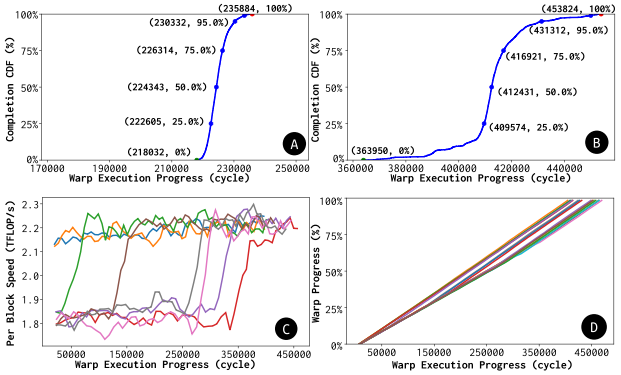


Figure 14: ❶ CDF of elapsed latency in exclusive blocks; ❷ CDF of elapsed latency in shared blocks; ❸ GFLOP/s distribution *w.r.t.* execution progress from left to right; ❹ Progress timeline of shared blocks, with slope denoting speed. Every warp in shared blocks first experiences a slow stage ($\sim$ 1.8 TFLOP/s) then jumps into a fast stage ($\sim$ 2.2 TFLOP/s).

experiences two stages: a *slower* stage at around 1.8 TFLOP/s followed by a *faster* stage at around 2.2 TFLOP/s till the termination. Moreover, by aligning the sampled timeline with start time and CU id, we identify that the transition from *slow* stage to *fast* stage is approximately when previously arrived shared block (executing at *fast* stage) terminates, suggesting an implicit FIFO-like prioritized scheduling policy. This finding explains the sparse-then-dense chaotic behavior in Fig. 11a and violates the common understanding of structured launch waves shown in Fig. 1A based on the algorithm [19]. Furthermore, we also identify similar effects on other kernels, *e.g.*, GEMM (Appendix J), with 50.93% tailing effect and jumping from $\sim$ 5 TFLOP/s to $\sim$ 7.5 TFLOP/s, spotlighting the prevalence of the tailing effect in shared blocks.

Regarding the performance, both exclusive and shared blocks are not optimal, yet have different performance statistics. Sharing blocks experience poor cache behavior due to the random-like memory access pattern, which can be verified by 5.85x higher stall cycles due to L1 miss from hardware profilers [55] in Tab. 4. Quite the opposite, exclusive blocks are so synchronized that they exhibit considerably peaked memory usage (Fig. 1B) and suffer from 4.47x higher exposed stall cycles due to memory bus busy and similarly, 1.45x more stall cycles for compute pipeline contention, both implying potential space for performance optimizations.

Table 4: Exposed stall cycles and reasons of FlashAttn-v2 under exclusive and shared blocks, collected from the Nsight Compute [58] PC Sampling.

Stall Reason	Exclusive Block	Shared Block
`long_scoreboard` (Waiting Global Memory)	41,055	268,941 (**5.85x**)
`mio_throttle` (Memory I/O High Pressure)	303,694 (**4.47x**)	68,005
`math_pipe_throttle` (Compute Unit Busy)	616,913 (**1.45x**)	425,026

8 Discussion and Future Work

NEUTRINO and GPU Scheduling: While we demonstrate the significance of NEUTRINO in revealing new insights from the runtime scheduling behavior in §4.6 and §7, current experiments are still far from completely understanding or reverse-engineering the GPU scheduling policies. This is because GPU scheduling is hardware-implemented, unlike OS schedulers like CFS or EEVDF [83,97], and is multi-level, including stream-, block-, and the finest warp-level instruction scheduling. Moreover, as a shared system, the behavior of the scheduler will be highly affected by runtime dynamics, which can also be reflected by the randomness of DMAT.

Impact of GPU Sharing: GPU sharing, *i.e.*, concurrently executing multiple kernels, is a practical solution to utilize the GPU. Such sharing can be intra-process via CUDA/HIP streams, inter-process via MPS (Multi-Process Service), or with resource isolation via MIG (Multi-Instance GPU). NEUTRINO is currently thread-local (*will block thread until kernel finishes*) and the scope is process-local (only profile the execution of the local process in MPS or MIG). We keep the impact of sharing as an interesting future direction.

Completeness of Probe Verification: We identified and prohibited three key factors of unsafe probes in §3.4, yet the current verification is not complete. ❶ There remain uncovered security factors, such as unreachable synchronization points that could pause programs. ❷ Current verification might be too strong, *e.g.*, `jmp` could be supported if the target is within the probe [24]. GPU-kernel verification itself remains an open research problem, and existing works only address several perspectives, such as synchronization [76] or data races [40]. Thus, we keep probe verification as future work.

Abnormal Speedup of Probed Kernel: As shown in Tab. 3, NEUTRINO's probed kernel might present better performance than the original kernel. Based on in-depth experiments (Appendix K), we attribute the speedup to the assembler optimization. In addition to translating assembly into machine code, modern assemblers also incorporate many optimizations, such as reordering instructions for better execution flow and merging reusable registers based on dependency tracking. Our analysis in Appendix K demonstrates that NEUTRINO probe's additional registers and instructions may change register dependencies and can lead to better execution flow (with 5.88% IPC improvement) and better performance (up to 0.94x faster). We believe this counterintuitive observation promises new opportunities since assemblers and machine code are not widely explored due to their hardware-oriented nature. For example, recently, DeepSeek's DeepGEMM [100] reported 10% speedup by flipping a control bit in machine code.

Towards Hardware-Software Profiler: As a pure-software profiling system primarily based on the assembly layer, NEUTRINO cannot profile unprogrammable hardware events such as the cache miss, although the DMAT trace can be used for cache simulation [98]. Furthermore, its profiling is based

on execution, so it is hard to trace stall cycles, *i.e.*, no instruction scheduled, but it can still help diagnose stall cycles caused by memory I/O. Despite these drawbacks, NEUTRINO can be an excellent complement to current hardware-dependent and kernel-exclusive kernel profilers by fusing the missing information on both sides. Moreover, from the big picture of observability, the multi-scale probing feature of NEUTRINO makes it an intermediate to bridge the gap between architecture-level hardware profilers [7, 55, 58] and application-level software profilers [12, 87]. Hence, we keep it as an exciting direction to integrate platform-specific hardware profilers and framework-specific software profilers to build a unified framework for GPU kernel profiling.

9 Related Work

GPU Hardware Profiler: Current GPU kernel profiling systems, such as NVIDIA NSight [58] / CUPTI [55] or AMD RGP [7] / GPA [5], are *hardware-dependent* that profiling features require corresponding hardware implementation support, such as performance counters like cache hit/miss rate. These features, though unique, are hard to adapt to new hardware, *e.g.*, async tensor core that makes utilization metric [81] less reliable as computation is offloaded from threads to tensor cores. Moreover, they cannot be flexibly customized to meet the needs of developers. For example, hardware profilers can only profile the entire program in a *sampling-based manner with low sampling frequency* to control overhead in performance and resources, limiting its capability to trace user-specified events. Instead, NEUTRINO limits the profiling targets to only the desired tracepoints, achieving both fine-grained event tracing and low system overhead.

GPU Software Profiler: Other framework-specific software profilers, such as the built-in profiler of PyTorch [87] and JAX [27], are *kernel-exclusive* and can only capture higher-level events, like measuring memory events (`alloc/free`) or timing the whole kernel (FLOP/s). Instead, NEUTRINO focuses on intra-kernel profiling at the instruction level.

GPU Micro-Benchmarking: Motivated to understand hardware design, GPU micro-benchmarking [1, 38, 51, 64, 85] tries to profile the *idealized performance* of specific hardware via *specially designed workloads* that only consist of interested instructions, such as `mma` to benchmark tensor core, without any other instructions (even those necessary in real workload, such as `ld` to read data) to reduce disturbance. Instead, NEUTRINO aims to measure the performance of real workloads, rather than idealized workloads.

GPU Simulation: Another way to understand the performance is to use simulators [9, 32, 42, 49, 92] that emulate GPU execution at the cycle level on a CPU. The major problems of these simulators lie in the speed of both running simulations (which might need several days) and support for new hardware features and instructions (which might take several years). Moreover, various runtime dynamics, *e.g.*, timing an instruction, may not be accurately profiled on simulators.

GPU Instrumentation: Binary instrumentation [15, 24, 50, 74] that injects *code, functions or interrupts*, with program states as parameters has been proven powerful in building performance tools on the CPU. There are also some GPU binary instrumentation tools in compile-time such as Ocelot [23], HIPAnalyzer [21], CUDAAdvisor [77], CUDAFlux [14], or in runtime such as SASSI [82], NVBit [93] and GTPin [80]. Though compiler-based approaches may benefit from additional information in compilation, they are bound to a specific compiler, module, or IR, lacking generalizability. They also require source codes, limiting its compatibility with existing frameworks. Runtime approaches directly operating on machine code lacks sufficient virtualization, so they mostly rely on the protection of the stack via injecting *pure device function*, which prohibits cooperation between probes for advanced usages, making them hard to perform tasks such as timing instruction (subtracting two clock readings) because the start time has been cleared on return and is not visible in the end timer's context. NEUTRINO persists runtime, rather than compiler, to advance its generalizability and targets parallel assemblies, other than machine code, to support more advanced and complex profiling tasks with cooperative probes.

10 Conclusion

The rapid development of AI systems has fostered an urgent need for comprehensive insights through advanced GPU kernel profiling. To address this, we present NEUTRINO, a GPU assembly probing infrastructure that enables fine-grained, versatile, and programmable GPU kernel runtime profiling with its distinct probe design of snippet, tracepoint, and map. We implement NEUTRINO, consisting of the hook driver, probe engine, and DSL compiler, for the CUDA and ROCm ecosystems. We introduce the novel Densified Memory Access Timeline (DMAT) to effectively visualize comprehensive GPU memory access patterns. We conduct extensive experiments, validating NEUTRINO's reliability, low overhead, and applicability. Additionally, we conduct a case study on the impacts of synchronization, successfully pinpointing performance bottlenecks with new insights gained by NEUTRINO. To maximize the potentials of NEUTRINO, we have open-sourced it at `https://github.com/open-neutrino/neutrino` and plan to build a collaborative community to support its continuous growth towards a unified framework for GPU kernel profiling.

Acknowledgement

We thank our shepherd, Xiaosong Ma, and the anonymous reviewers for their valuable feedback. We also thank Sheng Lyu, Manwen Liao, and Weiying Hou for the constructive discussion. This work is supported by NSFC Grant No. 62222216 and Hong Kong RGC ECS Grant No. 27204522.

References

[1] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed Badawy. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis, 2022.

[2] Chih-Chieh Yang Adnan Hoque, Less Wright. Deep Dive on the Hopper TMA Unit for FP8 GEMMs. https://pytorch.org/blog/hopper-tma-unit/, 2024.

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

[4] AMD. GCN Assembly. https://gpuopen.com/learn/amdgcn-assembly/, 2024.

[5] AMD. GPU Performance API. https://gpuopen.com/gpuperfapi/, 2024.

[6] AMD. HCC Compiler for ROCm. https://github.com/ROCm/hcc, 2024.

[7] AMD. Radeon GPU Profiler. https://gpuopen.com/rgp/, 2024.

[8] Jason Ansel et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024.

[9] Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L. Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. Navisim: A highly accurate gpu simulator for amd rdna gpus. In *the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 333–345, New York, NY, USA, 2023.

[10] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[11] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, June 1969.

[12] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with Scalene. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 51–64, Boston, MA, July 2023.

[13] bpftrace: High-level tracing language for Linux. https://github.com/bpftrace/bpftrace, 2025.

[14] Lorenz Braun and Holger Fröning. Cuda flux: A lightweight instruction profiler for cuda applications. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 73–81, 2019.

[15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 265–275, USA, 2003.

[16] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 178–191, New York, NY, USA, 2024.

[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: an automated end-to-end optimizing compiler for deep learning. In *the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018.

[18] Inho Cho, Seo Jin Park, Ahmed Saeed, Mohammad Alizadeh, and Adam Belay. LDB: An efficient latency profiling tool for multithreaded applications. In *21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI '24, pages 1497–1510, Santa Clara, CA, April 2024.

[19] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, ICLR '24, 2024.

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

[21] Sébastien Darche and Michel R. Dagenais. Low-overhead trace collection and profiling on gpu compute kernels. *ACM Trans. Parallel Comput.*, 11(2), June 2024.

[22] Peter J. Denning. Working set analytics. *ACM Comput. Surv.*, 53(6), February 2021.

[23] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 353–364, New York, NY, USA, 2010.

[24] eBPF. https://ebpf.io, 2024.

[25] Executable and Linkable Format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, 2024.

[26] Unified Acceleration Foundation. oneAPI Programming Model. https://oneapi.io/, 2025.

[27] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[28] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19, page 1069–1084, New York, NY, USA, 2019.

[29] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, and Abhinav Pandey et.al. The llama 3 herd of models, 2024.

[30] Khronos Group. SPIR: The Standard IR for Parallel Compute and Graphics. https://www.khronos.org/spir/, 2024.

[31] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024.

[32] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy Rogers. Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level. In *2018 IEEE International Symposium on High Performance Computer Architecture*, HPCA '18, pages 608–619, 2018.

[33] Troy D. Hanson and Arthur O'Dwyer. uthash. https://troydhanson.github.io/uthash/, 2024.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[35] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. In D. Song, M. Carbin, and T. Chen, editors, *the Machine Learning and Systems*, volume 5, pages 680–694. Curan, 2023.

[36] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 402–416, New York, NY, USA, 2022.

[37] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *the 29th Symposium on Operating Systems Principles*, SOSP '23, page 642–657, New York, NY, USA, 2023.

[38] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking, 2019.

[39] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO '98, pages 216–225, 1998.

[40] Aditya K. Kamath and Arkaprava Basu. iguard: In-gpu advanced race detection. In *the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 49–65, New York, NY, USA, 2021.

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

[42] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: an extensible simulation framework for validated gpu modeling. In *the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 473–486. IEEE Press, 2020.

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

[44] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004.

[45] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 2–14, 2021.

[46] Adnan Hoque Less Wright. Deep Dive on Cutlass Ping-Pong GEMM Kernel. https://pytorch.org/blog/cutlass-ping-pong-gemm-kernel/, 2024.

[47] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 663–679, Boston, MA, July 2023.

[48] Chang Liu, Byungchul Tak, and Long Wang. Understanding performance of ebpf maps. In *the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 9–15, New York, NY, USA, 2024.

[49] Changxi Liu, Yifan Sun, and Trevor E. Carlson. Photon: A fine-grained sampled simulation methodology for gpu workloads. In *the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1227–1241, New York, NY, USA, 2023.

[50] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005.

[51] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[52] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *the 29th Symposium on Operating Systems Principles*, SOSP '23, page 595–610, New York, NY, USA, 2023.

[53] NVIDIA. CUDA C++. hhttps://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024.

[54] NVIDIA. CUDA Compiler Driver NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, 2024.

[55] NVIDIA. Cuda profiling tools interface. https://docs.nvidia.com/cupti/index.html, 2024.

[56] NVIDIA. Fat Binaries. https://docs.nvidia.com/cuda/nvfatbin/index.html, 2024.

[57] NVIDIA. Instruction Set Reference. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-reference, 2024.

[58] NVIDIA. NSight Compute System. https://developer.nvidia.com/nsight-compute, 2024.

[59] NVIDIA. NVTX: NVIDIA Tools Extension SDK. https://github.com/NVIDIA/NVTX, 2025.

[60] NVIDIA. PTX ISA 8.5. https://docs.nvidia.com/cuda/parallel-thread-execution, 2024.

[61] NVIDIA. PTXAS. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, 2024.

[62] NVIDIA. TensorRT. https://developer.nvidia.com/tensorrt, 2024.

[63] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D. Owens. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu. In *the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 429–431, New York, NY, USA, 2023.

[64] M-M Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. Micro-benchmarking the gt200 gpu. *Computer Group, ECE, University of Toronto, Tech. Rep*, 2009.

[65] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and

Young ri Choi. HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference*, ATC '20, pages 307–321, July 2020.

[66] Mathias Payer, Enrico Kravina, and Thomas R. Gross. Lightweight memory tracing. In *2013 USENIX Annual Technical Conference*, ATC '13, pages 115–126, San Jose, CA, June 2013.

[67] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019.

[68] Tom Preston-Werner and Pradyun Gedam. TOML. https://toml.io/en/, 2024.

[69] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

[70] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[71] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[72] A. Roth and G.S. Sohi. Register integration: a simple and efficient implementation of squash reuse. In *the 33rd Annual IEEE/ACM International Symposium on Microarchitecture.*, MICRO '33, pages 223–234, 2000.

[73] Gabin Schieffer, Daniel Araújo De Medeiros, Jennifer Faj, Aniruddha Marathe, and Ivy Peng. On the rise of amd matrix cores: Performance, power efficiency, and programmability. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '24, pages 132–143, 2024.

[74] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with Bit-Precision. In *2005 USENIX Annual Technical Conference*, ATC '05, Anaheim, CA, April 2005.

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

[76] Rahul Sharma, Michael Bauer, and Alex Aiken. Verification of producer-consumer synchronization in gpu programs. *SIGPLAN Not.*, 50(6):88–98, June 2015.

[77] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *the 2018 International Symposium on Code Generation and Optimization*, CGO '18, page 214–227, New York, NY, USA, 2018.

[78] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 701–718, Boston, MA, July 2023.

[79] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[80] Alex Skaletsky, Konstantin Levit-Gurevich, Michael Berezalsky, Yulia Kuznetcova, and Hila Yakov. Flexible binary instrumentation framework to profile code running on intel gpus. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '22, pages 109–120, 2022.

[81] Benjamin Spector, Aaryan Singhal, Simran Arora, and Chris Re. GPUs Go Brrr. https://hazyresearch.stanford.edu/blog/2024-05-12-tk, 2024.

[82] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. Flexible software profiling of gpu architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 185–197, 2015.

[83] I. Stoica and H. Abdel-Wahab. Earliest eligible virtual deadline first : A flexible and accurate mechanism for proportional share resource allocation. Technical report, Old Dominion University, USA, 1995.

[84] Hao Sun and Zhendong Su. Validating the eBPF verifier via state embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '24, pages 615–628, Santa Clara, CA, July 2024.

[85] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):246–261, January 2023.

[86] LLVM Team. LLVM PTX Backend. https://llvm.org/docs/NVPTXUsage.html, 2024.

[87] PyTorch Team. PyTorch Profiler. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html, 2024.

[88] PyTorch Team. PyTorch Vectorized Elementwise Kernel. https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/cuda/CUDALoops.cuh, 2024.

[89] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[90] Dave Thaler. BPF Instruction Set Architecture (ISA). https://www.rfc-editor.org/info/rfc9669, 2024.

[91] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL '19, page 10–19, New York, NY, USA, 2019.

[92] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. Need for speed: Experiences building a trustworthy system-level gpu simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture*, HPCA '21, pages 868–880, 2021.

[93] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 372–383, New York, NY, USA, 2019.

[94] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, William Berman, Yiyi Xu, Steven Liu, and Thomas Wolf. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers, 2022.

[95] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[96] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020.

[97] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards achieving fairness in the linux scheduler. *SIGOPS Oper. Syst. Rev.*, 42(5):34–43, July 2008.

[98] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 191–208, 2020.

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

[100] Chenggang Zhao, Liang Zhao, Jiashi Li, and Zhean Xu. DeepGEMM: clean and efficient FP8 GEMM kernels with fine-grained scaling. https://github.com/deepseek-ai/DeepGEMM, 2025.

A Artifact Appendix

A.1 Abstract

The artifact of NEUTRINO is hosted at GitHub (in branch `artifact`), containing the source code, installation/collection/analysis scripts, collected traces that reproduce all the evaluation results in our paper. We also package the artifact evaluation as Jupyter Notebooks hosted on Google Colab, offering one-click results reproduction without local runtime setup. In addition, we also maintain an online documentation of NEUTRINO containing project highlights, user guides, roadmaps, and references for evaluating the functionality.

Artifact Claim: The collected traces and the codes are identical to our paper's corresponding description. You can replicate all the major results using the traces and analysis codes we provided (details in the *Expected Results* section below). We also provide the trace collection code for you to collect your own traces on your own devices. It's worth noticing that customized traces, particularly DMAT, can only yield similar results due to hardware and runtime dynamics.

A.2 Scope (meta-information)

- **Design**: NEUTRINO is a GPU assembly probing tool designed to *attach small snippets (probes)* to *GPU kernels at runtime* to expose runtime execution details (profiling).
- **System**: NEUTRINO system consists of two parts, the probe engine to attach code snippets, and the hook driver to capture GPU kernels launched at runtime. The source code is available at GitHub and is installable as a Python package.
- **Probes**: NEUTRINO probes are small TOML files that define the profiling task via snippet, datamodel, position, and callback. Probes used in the paper are available at Github.
- **Output**: Fig. 1, Fig. 10 (A/B/C), Fig. 11, Fig. 12, Fig. 13, and Table. 2 in the paper.
- **Evaluation**: We arrange evaluations in notebooks structured linearly, allowing simple click `Runtime -> Run All` execution. Please refer to the README, and instructions in each Jupyter Notebook (Colab).
- **Special Requirements**: No special requirements for static trace analysis, For dynamic trace collection, a NVIDIA GPU, e.g., A100, and a PTX-included build of PyTorch v2.5.0 and CUTLASS v3.5.0 are required.
- **Disk Space Requiremetns**: Evaluating on Google Colab doesn't require any disk space. Regarding local evaluation, please arrange 3GB for static traces and at least 10GB for collecting dynamic traces.
- **Experiment Time** Less than 30 minutes for **static** evaluations analyzing collected traces on CPU, and ≈ 10 hours for **dynamic** evaluation collecting traces on GPU.
- **Environment Setup Time**: For **static** evaluation, it takes ≈ 2 minutes to download traces. For **dynamic** evaluation, it takes ≈ 15 seconds to build NEUTRINO. Setting up PyTorch and CUTLASS might take ≈ 3 minutes.
- **Publicly Available**: Yes.
- **License**: We use the Apache License, Version 2.0 for the system source code and the CC BY 4.0 license for probes used in the paper.

A.3 Contents

NEUTRINO's artifact evaluation is arranged in 6 parts, corresponding to different figures or tables in the paper:

1. `block_sched`: §4.6
2. `dmat`: Fig. 1, Fig. 11
3. `kernel_overhead`: Tab. 3
4. `max_mem`: Fig. 12
5. `exposed_latency`: Fig. 13
6. `warp_sched`: Fig. 14

We arrange each part to correspond to a section in the Jupyter Notebook. Moreover, each evaluation is provided in two modes: the **static** that parses collected traces, suitable for *Getting Started* on local CPU-only devices without special hardware/software requirements, and the **dynamic** that collects the traces on the real GPU-enabled environment, suitable for *Full Evaluation*.

A.4 Hosting and Requirements

A.4.1 How to access

Please choose one of the following to access the artifact:

- **Github**:
  1. **Static** evaluation: `artifact/static.ipynb`
  2. **Dynamic** evaluation: `artifact/dynamic.ipynb`
- **Colab**:
  1. **Static** evaluation (Use CPU as Runtime) is here.
  2. **Dynamic** evaluation (Use GPU as Runtime) is here.

A.4.2 Hardware Requirement

For **static** evaluation, only a CPU machine with Python 3 runtime is needed. You *don't need to install* NEUTRINO for static evaluation.

For **dynamic** evaluation, you will need a NVIDIA GPU with the CUDA driver installed. Please note:

1. The choice of hardware will significantly affect results:
   - Please use RTX3080 for all DMAT plot (Part. 2).
   - Please use A100 for all the rest (Part. 1, Part. 3-6).
2. Please make sure no other workload is executing on the same GPU.
3. Please arrange enough disk space, at least 10GB, for dynamic traces collection.

A.4.3 Software Requirements

NEUTRINO system only depends on GNU toolchain (`gcc`, `file`, `git`, `nm`), CUDA toolchain (`cuobjdump`, `ptxas`) and

Python 3.12 (`pip`, `toml`). But evaluation workload needs a PTX-included build of PyTorch and CUTLASS. We package the dependency checking and installation in `prepare_env.py` for one-click installation.

A.4.4 Installation

It's recommended to use virtual environments, e.g., `conda create -y -n ae_env python=3.11 && conda activate ae_env`, for installation when not using Colab.

Automatic Installation:: We provide a helper script `prepare_env.py` that one can `python prepare_env.py` to install all dependencies and `neutrino`. Jupyter Notebooks (also Google Colab) use this way.

Manual Installation::

1. Clone the Github repository: `git clone -b artifact https://github.com/open-neutrino/neutrino.git`
2. Build and install neutrino: `cd neutrino && python setup.py install && cd ..`
3. Test installation with `neutrino -help`

Please refer to the README file for detailed descriptions on installing PTX-included builds of PyTorch and CUTLASS.

A.5 Evaluation Workflow

A.5.1 Getting Started Instructions

Getting started instructions, taking <30 min, consist of:

1. All **static** evaluation that reproduces all figures and tables in the paper based on collected traces.
2. The `block_sched` section (1st part) of the **dynamic** evaluation that collects and analyzes the block scheduling traces. This part takes <1 minute and helps justify the correct environment setup for detailed instructions.

You can use Colab to execute the evaluation scripts. To do this, first select the correct Runtime (CPU or GPU as stated above), then click the Runtime button at the top of the Colab web page, and click the Run All button in the dropdown menu to execute the scripts. Each section (of several blocks) can be executed independently. Statistics or figures will be displayed below each cell when execution finishes.

If you choose to evaluate locally, please download the Jupyter Notebooks and follow the same steps as the Colab execution instructions above.

A.5.2 Detailed Instructions

The detailed instructions cover the rest five sections of the **dynamic** evaluation. They are also packaged in a Jupyter Notebook (also available on Colab), allowing one-click (`Runtime -> Run All`) execution and evaluation. Each section can also be executed independently. So you can clear up traces after each section to save disk space.

A.5.3 Expected Results

Static evaluation on collected traces are expected to closely fit the figures and tables presented in the paper. To save disk space, we mistakenly deleted the original traces for these results. And because these results capture the finest runtime dynamics of the GPU, exact reproduction will be impossible. Our later experiments can only reproduce similar results. Please accept our apologies for the inconvenience, and we will update the revised paper to include the latest results.

Dynamic evaluation on customized traces is expected to produce similar results, i.e., similar numbers or figure shapes.

A.5.4 Further Evaluation

After completing the above evaluation and reading the documentation, we recommend several ways for further evaluation:

1. **Test your workloads**: NEUTRINO supports most GPU workloads. You can import your GPU kernels (CUDA C++, Triton, etc) and test them via `neutrino <your workload>`. Check more on NEUTRINO's support here.
2. **Test your workloads**: First, read the Programmable Probe guide, write and save your probe in `.toml` locally, and apply it using `neutrino -p <path/to/probe>`.
3. **Investigate Implementation**: NEUTRINO's implementation is well organized, and it's a good entry to understand how GPU code dispatches from OS. You can find the implementation of hook driver in `neutrino/src/` and the probe engine in `neutrino/probe/`.

A.5.5 Experiments Added in Shepherding

In the shepherding process, we have added several more experiments to address technical comments by reviewers. Though not required by AE, we also prepare the reproduction code:

- Microbenchmark (Tab. 2): `microbench.ipynb`
- Global DMAT (Fig. 15): `dmat_global.ipynb`
- DMAT Slowdown (Fig. 16): `dmat_slowdown.ipynb`
- Abnormal Speedup (Appendix K): `speedup.ipynb`

B Hook Driver

Hook driver is implemented via exposing functions of the same signature and using `dlfcn` to call the original function with hooks (lines 6 & 8) left for adding functionalities.

```
#define REAL_DRIVER ...   // path to real driver
static void* dlib = NULL; // dlopen handle
CUresult cuInit(unsigned Flags){ //same signature
  if (!dlib) {dlib = dlopen(REAL_DRIVER, RTLD_LAZY);}
  CUresult (*real)(unsigned) = dlsym(dlib, "cuInit");
  // insert code here := eBPF uprobe
  CUresult ret = real(Flags);
  // insert code here := eBPF uretprobe
  return ret; }
```

Hook driver is injected into the program via `LD_PRELOAD` like above for dynamic loading, *e.g.,* PyTorch, or `LD_LIBRARY` for static linking, *e.g.,* Triton. This can also be extended to filter out proprietary products like `cublas` for legal safety.

```
#define REAL_DRIVER ... // filled in make
#define HOOK_DRIVER ... // filled in make
void* dlopen(const char *filename, int flags) {
  // RTLD_NEXT -> the real dlopen of libc
  real_dlopen = dlsym(RTLD_NEXT, "dlopen");
  if (strstr(filename, "libcuda.so.1") != NULL) {
    void* tmp[STACK_TRACE_SIZE];
    int size = backtrace(tmp, STACK_TRACE_SIZE);
    char** syms = backtrace_symbols(tmp, size);
    for (int i = 0; i < size; i++) {
      // filter proprietaries like cublas
      if (strstr(syms[i], "cublas") != NULL)
        return dlopen(REAL_DRIVER, flags);
    }
    return dlopen(HOOK_DRIVER, flags);
  }
  return dlopen(filename, flags);
}
```

C Probe Map Planning

Our probe engine will automatically plan the probe map according to the definition, *e.g.,* `thread:array:8:1`. Its mechanism is similar to the following CUDA/HIP C++:

```
#define NO_BYTES ... // filled by datamodel, 8*1 here
__global__ void tmp(void* buff) {
  int thr_idx = (blockDim.y * threadIdx.z + \
    threadIdx.y) * blockDim.x + threadIdx.x;
  int blk_idx = (gridDim.y * blockIdx.z + \
    blockIdx.y) * gridDim.x + blockIdx.x;
  int blk_size = blockDim.x * blockDim.y * blockDim.z;
  int buf_idx = blk_idx * blk_size + thr_idx;
  void* buf_loc = buff + buf_idx * NO_BYTES;
}
```

In practice, the probe engine achieves it via formatting a Python string of assemblies with `name` and `no_bytes` like the following example of PTX:

```
""".reg .b32  %loc<7>; // applies to all map
mad.lo.s32  %loc7, %ntid.y, %tid.z, %tid.y;
mad.lo.s32  %loc6, %loc7, %ntid.x, %tid.x;
mad.lo.s32  %loc5, %nctaid.y, %ctaid.z, %ctaid.y;
mad.lo.s32  %loc4, %loc5, %nctaid.x, %ctaid.x;
mul.lo.s32  %loc3, %ntid.x, %ntid.y;
mul.lo.s32  %loc2, %loc3, %ntid.z;
mad.lo.s32  %loc1, %loc2, %loc4, %loc5;
// following is unique for each map
.reg .b64 %map_{name}<5>;
mul.wide.s32  %map_{name}4, %loc1, {no_bytes};
ld.param.u64  %map_{name}3, [param_{name}];
cvta.to.global.u64  %map_{name}2, %map_{name}3;
add.s64 %map_{name}1, %map_{name}2, %map_{name}4;"""
```

D Probe DSL Compilation Example

Our compiler will first translate the DSL into eBPF-like IR:

```
callback="block_sched.py"
[map.block_sched]
type  = "array"
level = "warp"
size  = "16"
cap   = "1"
[probe.thread_start_thread_end]
position = "kernel"
level = "warp"
register = {"u32": 2, "u64": 3}
before = "clock PD0"
after = """clock PD1
sub PD1, PD0
cvt32 P0, PD1
cuid P1
SAVE [block_sched] {PD0, P0, P1}"""
```

This intermediate representation is inspired by the mature eBPF ISA [90] and can be translated into platform-specific assemblies like PTX:

```
[probe.thread_start_thread_end]
... # other metadata truncated
before = """.reg .b64 %PD<3>;
.reg .b32 %P<2>;
mov.u64 %PD0, %clock64;"""
after = """mov.u64 %PD1, %clock64;
sub.u64 %PD1, %PD1, %PD0;
cvt.u32.u64 %P1, %PD1;
mov.u32 %P2, %smid;
SAVE [block_sched] {%PD0, %P1, %P2};"""
```

and GCN Asm:

```
[probe.thread_start_thread_end]
... # other metadata truncated
register={"u32": 2, "u64": 3, "type": "sgpr"}
before = "s_memrealtime PD0"
after = """s_memrealtime PD1
SUB64 PD1, PD1, PD0
CVT32 P0, PD1
s_getreg_b32 P1, hwreg(HW_REG_HW_ID)
SAVE [block_sched] {PD0, P0, P1}"""
```

On AMD GPUs, scalar registers per warp and vector registers per thread are separated, which is reflected in the new `type` field in `reigster`. We also implement more helpers for AMD platform to cover the missing functionality, such as `SUB64` for 64-bit subtraction as AMD doesn't support 64bit add/sub for Scalar Registers [4] (We emulate it by `s_sub_u32`).

E Trace Analysis Code Example

Typical NEUTRINO trace analysis code consists of two parts: ❶ *(before line 32)* Trace reading code auto-generated by NEUTRINO probe engine, for reading binary traces into Python data structures via `struct`. ❷ *(after line 32)* Trace analysis code built upon read structures. The whole program can be registered to the hook driver as `CALLBACK` (Fig. 10), like the following example:

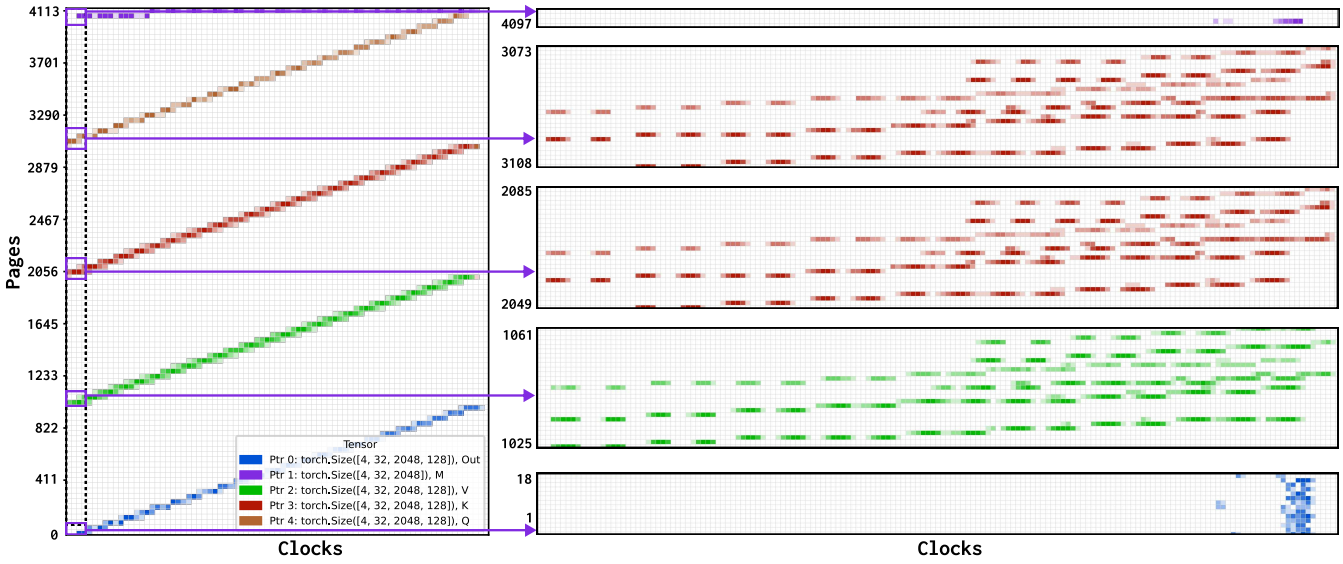


Figure 15: NEUTRINO DMAT aligned to the GPU-local clock. Left: The overview; Right: the Zoom-in view of selected small proportions highlighted in the red rectangles.

```
# Neutrino Generated Code for Reading Trace
import struct
from typing import NamedTuple, List, Tuple
from neutrino import TraceHeader, TraceSection
class block_sched(NamedTuple):
  start: int
  elapsed: int
  cuid: int
def parse(path: str):
  with open(path, "rb") as f:
    header: TraceHeader = TraceHeader(
      struct.unpack("iiiiiiii", f.read(32)))
    sections: List[TraceSection] = []
    for _ in range(header.numProbes):
      size, offset = struct.unpack("QQ", f.read(16))
      sections.append(TraceSection(size, offset))
    gridSize = header.gridDimX * header.gridDimY
                               * header.gridDimZ
    blockSize = header.blockDimX * header.blockDimY
                                 * header.blockDimZ
    records: List[List[block_sched]] = []
    for i in range(gridSize):
      records.append([])
      for j in range(blockSize):
        start, elapsed, cuid = struct.unpack(
                                 "QII", f.read(16))
        records[i].append(
                 block_sched(start, elapsed, cuid))
  return header, sections, records
# END OF GENERATED CODE
import numpy as np
header, sections, records = parse(sys.argv[1])
unique_cus = set()
for block in records:
  unique_cus.add(block[0].cuid)
cu_timelines = [[]] * len(unique_cus)
sched_times = [0.0] * len(unique_cus)
work_times = [0.0] * len(unique_cus)
for cur in records:
  sched_out = False
  for block in cu_timelines[cur[0].cuid]:
    if block.start+block.elapsed<=cur[0].start:
      sched_times[cur[0].cuid]+=cur[0].start
                 - (block.start + block.elapsed)
      cu_timelines[cur[0].cuid].remove(block)
      cu_timelines[cur[0].cuid].append(cur[0])
      work_times[cur[0].cuid] += cur[0].elapsed
      sched_out = True
      break
    if not sched_out:
      cu_timelines[cur[0].cuid].append(cur[0])
      work_times[cur[0].cuid] += cur[0].elapsed
print(np.array(sched_times).mean(),
      np.array(work_times).mean())
```

F Persistent Kernel for `torch.zeros`

Following code is generated by GPT4o with prompt: "Write a Persistent Kernel to initialize a torch tensor to 0":

```
import torch
import triton
import triton.language as tl
@triton.jit
def zero_persistent_kernel(output_ptr, numel,
  BLOCK_SIZE: tl.constexpr, NUM_SMS: tl.constexpr):
  start_pid = tl.program_id(axis=0)
  num_blocks = tl.cdiv(numel, BLOCK_SIZE)
```

```
  blocks_per_sm = num_blocks // NUM_SMS
  if start_pid < num_blocks % NUM_SMS:
    blocks_per_sm += 1
  block_id = start_pid - NUM_SMS
  for _ in range(blocks_per_sm):
    block_id += NUM_SMS
    offsets=block_id*BLOCK_SIZE+tl.arange(0,BLOCK_SIZE)
    mask = offsets < numel
    tl.store(output_ptr + offsets,
      tl.zeros([BLOCK_SIZE], dtype=tl.float16), mask)
def zero_persistent(x: torch.Tensor):
  numel = x.numel()
  NUM_SMS = torch.cuda.get_device_properties("cuda")\
                     .multi_processor_count
  BLOCK_SIZE = 128
  grid = lambda META: (min(NUM_SMS,
     triton.cdiv(numel, META['BLOCK_SIZE'])),)
  zero_persistent_kernel[grid](
    x, numel, BLOCK_SIZE, NUM_SMS)
t=torch.empty((4096,4096),torch.float16,device="cuda")
zero_persistent(t)
```

G Global Timer Aligned DMAT

NEUTRINO DMAT can also be synchronized to the GPU-local timer, *i.e.*, different compute units share the same synchronized timer, rather than CU-local cycle timer. DMAT of the same configuration as Fig. 1 under global synchronization becomes Fig. 15. Under global synchronization, DMAT becomes *terraced*-like and each *"step"* represents a launch wave of virtual blocks, 32x128 in this case, to physical compute units (108 SM on A100). We recommend zooming in on each *"step"*, for insightful analysis.

The zoomed-in view (right part of Fig. 15, note the y-axis value difference) reflects the actual memory access pattern applied to the hardware, and is insightful for hardware-level analysis and cache simulation.

H Micro-benchmarking Details

Here we present more details behind the theoretical estimation, and the reasons for the varying clock estimation. Take the most complicated `random` access as an example, we use the Fisher–Yates shuffle to create reproducible random index sequences in A. At runtime, each thread reads the random index from A and writes to B accordingly, like the following:

```
__global__ void random_kernel(
  const int* A, int* B, // A has been shuffled
  int M, int N, unsigned int NS) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if (col >= M) return; // boundary check
  int* src = A + col;
  for (int i = 0; i < M * N; i += M) {
    int idx  = src[i];  // get the random idx from A
    B[idx] = i;         // write to B[idx]
    device_sleep(NS);   // spinloop inside
  }
}
```

Then, in verification, we reconstruct the same random index and simulate the access as follows. We compare simulated addresses with collected addresses aligned to the base address.

```
arr = fisher_yates_shuffle(...)
for blockIdx in range(blocks):
  for threadIdx in range(threads):
    start = blockIdx * threads + threadIdx
    clock = LATENCY # 1st access doesn't sleep
    for recordIdx in range(records):
      index  = start + recordIdx * blocks * threads
      addr_A = (index * size) # >> 16 => page
      addr_B = (arr[index] * size) # >> 16 => page
      clock += NS + LATENCY
```

However, one critical challenge is the memory access latency (`LATENCY` in the above code), which, in practice, will vary from cache hit/miss. To be fair, we disable the L1 (via `cg` cache operator [60]) and choose the L1 disabled latency (570 cycles on A100) as `LATENCY`. Because different patterns have varying L1 hit rates (88% for linear and 0% for stride), resulting in unpredictable estimation errors.

I Evaluating NEUTRINO probe slowdown

As presented in Tab. 3, NEUTRINO's `dmat` probe leads to different degrees of slowdowns on different kernels, from 2.75x on Flash-Attn-v2 to 13.163x on SoftMax. We first analyze this effect from the perspective of memory I/O by sampling the DMAT access, *i.e.*, adding a sampling counter to make DMAT save only 1/2 (save only once in two memory accesses), 1/3, 1/4, 1/8, and measuring the kernel slowdown.

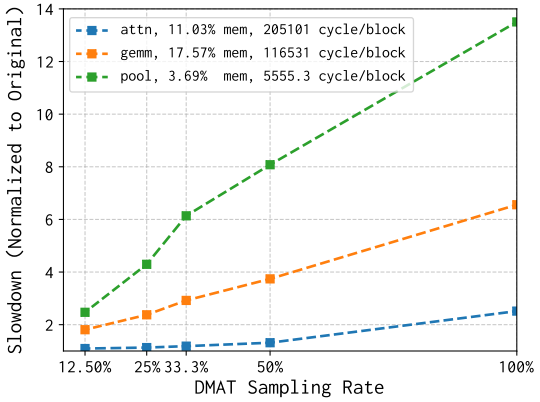


Figure 16: NEUTRINO DMAT Slowdown versus kernels and sampling frequency.

Results presented in Fig. 16 demonstrate that the slowdown of DMAT is mainly of a linear relationship *w.r.t.* the sampling

frequency, highlighting that most DMAT cost comes from the I/O. Moreover, we also identify two major factors contributing to the exact slowdown ratio. First, smaller kernels, *i.e.*, lower block time, suffer from larger slowdown as the proportion of `dmat`'s memory I/O would be larger *w.r.t.* the original block time (pool, ∼5555 cycles with 13x slowdown and attn, ∼205101 cycles with 2.75x slowdown). Second, for kernels of similar block time, the proportion of memory instructions significantly affects the slowdown (attn, 11.03% memory instructions with 2.75x slowdown, and gemm, 17.57% memory instructions with 6.55x slowdown).

J Case Study on GEMM Kernel

Here, we also formulate two configurations of GEMM kernel (`M=N=K=4096`) issuing the same burden to the physical SM: ❶ 128x256x64 tile, 3 stages, 8 warps, 1 block per SM on A100; ❷ 128x128x64 tile, 3 stages, 4 warps, 2 blocks per SM on A100. And we present the cumulative distribution function of block completion time in Fig. 17A and Fig. 17B, respectively.

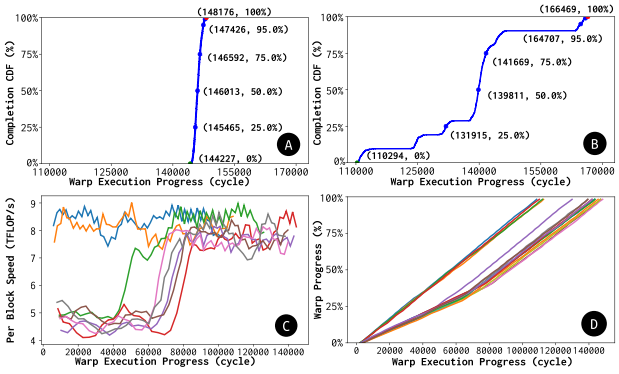


Figure 17: ❹ CDF of elapsed latency in exclusive blocks; ❺ CDF of elapsed latency in shared blocks; ❻ GFLOP/s distribution *w.r.t.* execution progress from left to right; ❼ Progress timeline of shared blocks.

We can identify the same pattern as §7 that shared blocks suffered from tailing blocks. Similarly, we plot the intra-block-level throughput timeline and the warp execution timeline in Fig. 17A and Fig. 17B, yielding similar finding as §7. But different from the previous, here `BLOCK_N`, 128 or 256, is a parallel dimension. Fig. 14 and Fig. 17 together demonstrate that the existence of tailing effects in shared blocks due to GPU synchronization and scheduling (§7) is general, regardless of kernels or tile organizations (parallel or sequential).

K Analyzing Abnormal Speedup

As shown in Tab. 3, NEUTRINO probes can lead to abnormal speedup. One significant case of such speedup is GEMM with `M=N=K=2048`, where `block_sched` can lead to up to 0.94x speedup on CUTLASS implementation and 0.96x speedup on Triton implementation. We pick the Triton implementation (`BLOCK_M=128`, `BLOCK_N=128`, `BLOCK_K=32`, `STAGES=3`, `WARPS=4`), and `block_sched` probe as it has the lowest interruption to the program (only thread start and end). We profiled the probed and original kernel (with the same `-O3`) with hardware profiler Nsight Compute [58] to validate their performance metrics, and the results suggest dramatic changes:

Metric	**Probed**	**Original**
`Executed IPC Elapsed`	1.08 (**+5.88%**)	1.02
`Issue Slots Busy`	30.41% (**+4.97%**)	28.97%
`long_scoreboard` (waiting global memory)	2,698,349 (**-10.22%**)	3,005,844
`mio_throttle` (waiting memory inst queue)	665,885 (**-32.8%**)	990,653
`no_instruction` (waiting instruction/register)	52,804 (**-17.32%**)	63,868
`short_scoreboard` (waiting shared memory)	1,311,106 (**+148%**)	528,495

Profiled metrics highlight a 5.88% IPC improvement with a 4.97% busier instruction scheduling. Moreover, we also identified that NEUTRINO probed kernel has 10.22% less waiting time for global memory, 32.8% less waiting time for memory instruction queue, and 17.32% less waiting time for instructions or register cache. To further address the improvement, we checked the assembled machine code, particularly the main loop with tensor core (`HMMA`) and async copy (`LDGSTS`):

```
/* Probed SASS */
LDGSTS.E.BYPASS ...
IMAD R132, R236 ...
HMMA.16816.F32  ...
HMMA.16816.F32  ...
HMMA.16816.F32  ...
IMAD.MOV.U32    ...
HMMA.16816.F32  ...
IMAD.MOV.U32    ...
IMAD R133, R236 ...
HMMA.16816.F32  ...
LDGSTS.E.BYPASS ...
HMMA.16816.F32  ...
HMMA.16816.F32  ...
HMMA.16816.F32  ...
IMAD R132, R236 ...
IADD3 R134, P0, ...
IMAD.MOV.U32    ...
HMMA.16816.F32  ...
```

```
/* Original SASS */
LDGSTS.E.BYPASS ...
HMMA.16816.F32  ...
LDGSTS.E.BYPASS ...
HMMA.16816.F32  ...
LDGSTS.E.BYPASS ...
HMMA.16816.F32  ...
IMAD.MOV.U32    ...
HMMA.16816.F32  ...
IMAD.MOV.U32    ...
HMMA.16816.F32  ...
LDGSTS.E.BYPASS ...
IMAD.X R251,    ...
ISETP.LE.AND P5 ...
HMMA.16816.F32  ...
LDGDEPBAR ;     ...
IADD3 R138,     ...
HMMA.16816.F32  ...
LDGSTS.E.BYPASS ...
```

From the above truncated and optimized machine code, we can find the probed kernel's `HMMA` instructions are more continuous, with more opportunities to reuse the register cache (as results are accumulated along `K`). Moreover, memory instructions `LDGSTS` are more distributed, possibly leading to a more balanced pressure on the memory system.

This experiment shows that new instructions from NEUTRINO probe(s) may lead the assembler [61] to better execution flow. Though the current experiment is not comprehensive (only one kernel), we believe that this abnormal speedup and our analysis spotlight there are still unexplored opportunities in low-level assembler optimization.

Principles and Methodologies for Serial Performance Optimization

Sujin Park Mingyu Guan Xiang Cheng Taesoo Kim
Georgia Institute of Technology

Abstract

Throughout the history of computer science, optimizing existing systems to achieve higher performance has been a longstanding aspiration. While the primary emphasis of this endeavor lies in reducing latency and increasing throughput, these two are closely intertwined, and answering the *how* question has remained a challenge, often relying on intuition and experience.

This paper introduces a systematic approach to optimizing sequential tasks, which are fundamental for overall performance. We define three principles—task removal, replacement, and reordering—and distill them into eight actionable methodologies: batching, caching, precomputing, deferring, relaxation, contextualization, hardware specialization, and layering. Our review of OSDI and SOSP papers over the past decade shows that these techniques, when taken together, comprehensively account for the observed sequential optimization strategies.

To illustrate the framework's practical value, we present two case studies: one on file and storage systems, and another analyzing kernel synchronization to uncover missed optimization opportunities. Furthermore, we introduce SysGPT, a fine-tuned GPT model trained on curated literature analysis, which offer context-aware performance suggestions. SysGPT's outputs are more specific and feasible than GPT-4's, aligning with core strategies from recent research without direct exposure, demonstrating its utility as an optimization assistant.

1 Introduction

Improving the performance of computer systems has been a pivotal objective within the field of computer science, profoundly influencing user experience, operational efficiency, and cost reduction. High-performance systems process tasks with greater speed and efficiency, thereby minimizing response times for user applications and enhancing the throughput of computational operations.

The optimization of sequential execution is particularly crucial, as Amdahl's law [8] highlights that the maximum potential speedup of a system is constrained by the fraction of the program that must remain sequential. Despite advances in parallel processing, addressing the sequential bottleneck remains foundational for achieving meaningful performance gains.

In response to this challenge, this paper proposes a structured approach to optimizing *sequential* performance. Our approach is founded on three key principles: the removal, replacement, or reordering of tasks within a sequence. The performance of sequential execution is inherently defined by the sequence itself, as each task's execution time directly contributes to the total runtime, and tasks must be completed in a specific order. This deterministic nature implies that modifying the sequence—either by removing tasks to reduce the overall sequence length, replacing slower tasks with faster alternatives, or reordering tasks for more efficient execution—constitutes the only practical means to improve sequential performance under a fixed hardware and execution environment.

Building upon these principles, we define eight actionable methodologies for sequential performance optimization: batching, caching, precomputing, deferring, relaxation, contextualization, hardware specialization, and layering. Each methodology can be systematically explained through one or more of the three fundamental principles, providing clear reasoning as to why and how it improves sequential performance.

While each methodology may be individually familiar to the systems community, our key contribution lies in synthesizing them into a unified framework that captures the full space of observed optimization opportunities for sequential execution. By distilling common optimization patterns into a systematic and actionable set of methodologies, our work clarifies the space of existing optimization opportunities and transforms the inherently open-ended challenge of performance optimization into a fully structured and tractable process.

To investigate the completeness of this framework, we conducted an empirical analysis by reviewing every system paper published at OSDI and SOSP over the past decade. Our goal was to examine whether the eight methodologies could sufficiently account for the diverse optimization techniques proposed in these papers, thereby demonstrating that the methodologies form a complete set of common patterns for sequential performance optimization. These eight methodologies are not intended to represent the *only* means of achieving sequential performance optimization, but rather to provide a comprehensive account of the common patterns observed over the past decade.

We envision these methodologies serving as a checklist for researchers and developers, helping them to systematically explore known optimization strategies, reduce the risk of overlooking opportunities, and recognize when a true algorithmic breakthrough may be required.

Finally, to bridge the gap between conceptual principles and practical application, we present SysGPT, a fine-tuned GPT model [2] trained on a curated analysis of a decade's literature. Through both qualitative and quantitative evaluation, we demonstrate that SysGPT consistently outperforms the base model and few-shot learning approaches. Our qualitative analysis shows that SysGPT generates more specific and actionable suggestions that better align with ground-truth solutions, while quantitative results demonstrate significant improvements in precision, recall, and F1-score across various temperature settings and sampling configurations. This combination of conceptual rigor and practical utility ensures that the insights gleaned from our analysis do not remain only theoretical, but instead become a tangible tool for improving system performance.

In summary, this paper has the following contributions:

- Formalizes the performance of sequential execution and defines three principles to optimize it. (§2)
- Establishes eight methodologies that systematically guide sequential performance improvements.(§3)
- Empirically validates the completeness of the framework through a comprehensive analysis of OSDI and SOSP papers over the past decade. (§3)
- Demonstrates the potential of AI-assisted performance optimization through the development of SysGPT. (§5)

2 Problem Definition

$$S_{parallel} = \frac{1}{F_{serial} + (F_{parallel}/\#\text{ threads})} \tag{1}$$

Optimizing sequential performance is a fundamental challenge in system design. Amdahl's law (Equation 1)[8] provides an insight into the potential speedup achievable through the utilization of multiple processors in a computing task. The law states that the maximum possible speedup from parallelizing a program ($S_{parallel}$) is limited by the fraction of the program that must be executed sequentially, denoted as the serial fraction (F_{serial}). In other words, it highlights the fact that not all parts of a program can be parallelized, so even with infinite additional resources, there will always be a limit to how much faster a program can run. It's obvious that augmenting $F_{parallel}$ (and diminishing F_{serial}) leads to better performance. However, there has been a lack of systematic approach to optimize F_{serial} of a program. Moreover, many problems (programs) cannot be parallelized due to computational dependencies among constituent tasks, making the optimization of F_{serial} the only way to accelerate programs.

$$S_n = \{t_i\}_{i=1}^{n} = \text{Sequence of tasks for one } \textit{epoch} \tag{2}$$

The serial fraction of a program plays a pivotal role in determining the overall runtime and is of significant importance in optimizing performance. We define the serial fraction of a program (F_{serial}) as a sequence of tasks, t_1, t_2, $\ldots t_n$ as represented in Equation 2.

$$latency = F(S_n) = F(\{t_i\}_{i=1}^{n}) \tag{3}$$

$$throughput = N, \text{ where } N \cdot F(S_n) < time \tag{4}$$

In many common scenarios, a sequence of tasks is executed iteratively, with the optimization of repetitive tasks being the primary focus for performance enhancement. Each iteration is referred to as an *epoch*, and Equation 3 signifies the elapsed time to execute one epoch, typically denoting *latency*. S_n represents the sequence of tasks defined above, and F represents the execution time required to process S_n, where n is the number of tasks (length of S_n). Equation 4 defines *throughput*, providing insight into the number of epochs that can be executed within a given time frame.

2.1 Principles for performance optimization

In pursuit of enhancing system performance for sequential workloads, the focus lies on optimizing the sequence of tasks, S_n, to ultimately reduce the runtime $F(S_n)$. Given that the optimization target is sequential, $F(S_n)$ is deterministically influenced by the sequence S_n and the hardware executing it. Aside from completely rewriting S_n, there are three strategic approaches: removal, replacement, or reordering of tasks within the sequence. The following three principles delineate these strategies:

- $\mathbf{P}_{rm}$**:** Remove t_i from S_n. Upon optimization, the length of the optimized sequence S'_m is shorter than the original sequence S_n, thereby satisfying $m < n$.
- $\mathbf{P}_{rep}$**:** Replace t_i with t_j within S_n. While the length of the sequence S_n may remain unchanged, the optimized sequence S'_n yields a better overall runtime performance, denoted as $F(S_n) > F(S'_n)$.
- $\mathbf{P}_{ord}$**:** Explore alternative permutations of S_n. Different permutations of S_n indicates different execution order of given tasks, which can impact overall runtime.

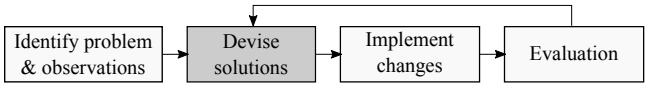


Figure 1: Steps in performance optimization.

2.2 Research scope

Our research targets optimizing the sequential fraction of system performance, focusing specifically on throughput and latency. Throughput reflects the system's capacity to process tasks within a given timeframe, such as request handling rates, model trainig speed, or data scanning speed. Latency measures responsiveness, including per-request or per-operation time, and offers insights into real-time performance and user experience. We focus solely on optimizing existing systems, excluding concerns such as security, energy use, space efficiency, fault tolerance, and maintainability. We also exclude the design of entirely new algorithms, as our goal is to optimize existing ones.

2.3 Steps in performance optimization

Performance optimization is a multi-step, iterative process involving inefficiency identification, solution design, implementation, and evaluation. While tools and benchmarks for problem identification and evaluation have matured over time, the intermediate steps—designing and implementing solutions—remain largely reliant on the researcher's expertise, creativity, and experience. These stages are often the most challenging, as they require both deep system knowledge and the ability to predict how changes will interact with the broader system dynamics. Within this context, this paper focuses on establishing a systematic approach for the second step: *devising solutions*.

3 The Eight Optimization Methodologies

The landscape of system optimization techniques is vast and often discussed informally, with little consensus on how many distinct strategies exist or how they relate to each other. In this section, we distill a decade of performance optimization research in the systems community and define eight actionable methodologies that together form a structured framework for sequential performance optimization: batching (§3.1), caching (§3.2), precomputing (§3.3), deferring (§3.4), relaxation (§3.5), contextualization (§3.6), hardware specialization (§3.7), and layering (§3.8). We explain each methodology through the principles of removal, replacement, and reordering, detailing its rationale and impact on sequential performance, and illustrating it with examples from representative system papers.

To empirically validate the coverage of our framework, we conducted a exhaustive review of 477 papers published at OSDI and SOSP over the past decade. Our analysis indicates that these eight methodologies collectively encompass all common patterns of sequential performance improvements observed in practice over the last ten years.

3.1 Batching

When the cost for computing a batch of items is lower than the total cost for computing each item individually, combining these items into batches will enhance overall efficiency. As shown in Table 1 (a), for every epoch, duplicate costs (x_2, x_4 and x_6) are incurred across epochs, and these costs are repetitively accrued with each iteration of epochs. With batching, depicted in Table 1 (b), these individual blocks are merged into one big block (x_7). To ensure the efficacy of batching, two conditions must be satisfied: the length of the optimized sequence should be shorter than that of the original sequence, and the running time for batched items should be shorter.

In terms of the three principles (§2.1), batching is a methodology which can effectively leverage all the three principles. First, batching commonly yields benefits through coalesced computation. By coalescing duplicate tasks together, particularly when each task incurs significant overhead, the overall runtime can be reduced. Upon comparing of the original and batched sequences, the latter removes x_2 and x_4 (P_{rm}), and replaces x_6 with x_7 (P_{rep}). Alongside P_{rm} and P_{rep}, the overall runtime improves. NEVE (①) is an example of this approach (see Table 2 for details).

Batching also inherently defers earlier tasks until a batch of tasks is gathered. Consequently, it has the potential to decrease the overall number of tasks to be processed by discarding stale tasks at the time of batched request (P_{rm}), thereby reducing the overall runtime. For instance, group commit and write buffer batch multiple updates and apply modifications in a deferred and batched manner, thereby some stale data that has already been deleted or updated does not need to be committed anymore. EAW (②) implements this approach by enabling applications to update batched items in the log prior to commit.

In addition, when batching is applied, the order of tasks is reordered (P_{ord}). For example, initially x_2 was located between x_1 and x_3, but after optimization, x_1 and x_3 will be executed consecutively. Therefore, batching might be employed to maximize the spatial and temporal locality of data. IX (③) efficiently adopts this strategy to batch requests at every stage of the network stack.

This concept is also closely interwined with caching (§3.2) as batching eventually provides greater caching opportunities with the improved locality.

In general, batched items can achieve higher bandwidth and throughput by eliminating redundant costs across tasks. Nevertheless, this approach can lead to extended latency, as initial earlier tasks in a batch might wait for subsequent tasks to arrive, and processing a batch usually takes longer than processing an individual task.

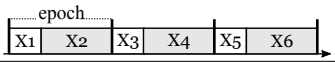
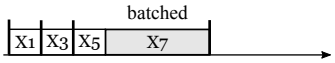
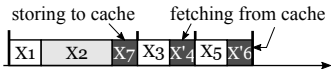
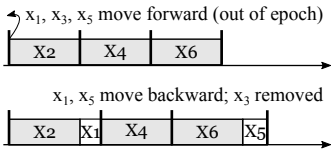
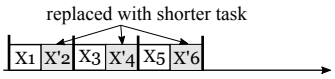
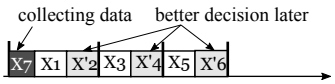
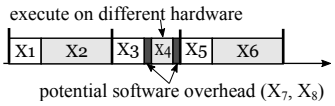
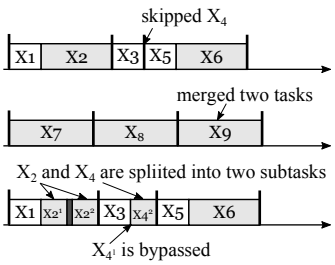
Methodology	P_{rm}	P_{rep}	P_{ord}	Visualization	Condition and Strategies
(a) Original					• Use below methodologies to reduce overall runtime.
(b) Batching	✓	✓	✓		$len(S_{before}) > len(S_{after})$ **AND** $F(x_2)+F(x_4)+F(x_6) > F(x_7)$ • Coalesce duplicate tasks together. • Discard stale tasks at the time of batched request. • Maximize spatial and temporal locality.
(c) Caching		✓			$F(x_4)+F(x_6) > F(x_7)+F(x_4')+F(x_6')$ • Introduce new caching layer on top of existing system. • Modify system to reveal more caching opportunities. • Develop caching policy to decide what to cache.
(d) Precomputing & Deferring	✓		✓		$len(S_{before}) > len(S_{after})$ **OR** $F(x_1,x_2) > F(x_2,x_1)$ • Move tasks out of a sequence and use resources in other time. • Reorder tasks within a sequence and seek for better locality. • Precomputing speculates a task will be required in near future. • Precomputing speculates future context. • Deferring expects the task will be shorter or not used at all. • Deferring creates an opportunity to exploit future contextual advantages such as caching or batching.
(e) Relaxation	✓	✓			$len(S_{before}) > len(S_{after})$ **OR** $F(x_k) > F(x_k')$ • Compromises other properties such as accuracy, consistency or durability. • Sampling is a common example of compromising accuracy.
(f) Contextualization		✓			$F(x_2)+F(x_4)+F(x_6) > F(x_7)+F(x_2')+F(x_4')+F(x_6')$ • Collect runtime context and make workload-specific decision. • Analyzing runtime characteristics can be done off the critical path then no need to pay extra cost.
(g) Hardware specialization		✓			$F(x_4) > F(x_7)+F'(x_4)+F(x_8)$ • Decide which tasks to execute on which hardware. • Tailored design for specific hardware.
(h) Layering	✓	✓	✓		Bypassing: $len(S_{before}) > len(S_{after})$ • Skip existing layer and take a fast-path. • The most common strategy is kernel bypass. Delayering: $len(S_{before}) > len(S_{after})$ **AND** $F(x_1)+F(x_2) > F(x_7)$ • Merge two tasks into one; singularity appraoch. Decoupling: $F(x_2) > F(x_2^1)+F(x_2^2)$ • Split one task into several fine-grained tasks. • Enable finer control and loosen dependencies. • Open more opportunities for other methodologies.

Table 1: Summary of the eight methodologies, detailing their underlying principles, visual representations, necessary conditions, and strategic applications.

3.2 Caching

While batching removes redundant costs across tasks, caching addresses redundant costs over time. The fundamental idea behind caching is to keep computational results in memory and reuse them at a later time, thereby avoiding redundant computation and leveraging previously computed values. Table 1 (c) illustrates how caching optimizes costs over time. For caching to be effective, the computational redundancy (x_4 and x_6) must outweigh the overhead of using and maintaining the cache. This overhead involves the cost for storing computed results in the cache (x_7) and retrieving old results from it (x_4' and x_6'). In other words, caching leverages P_{rep} by replacing redundant computation with cached results. Consequently, caching inherently reduces the overall runtime at the expense of increased memory usage.

Introducing a new caching layer on top of existing system involves identifying common operations that can be strategically cached for enhanced performance. Table 2 shows such examples to optimize stream processing (④) and VM allocation (⑤). Beyond introducing additional caching layer to the current system, it is also possible to reveal more caching opportunities by modifying existing structures (⑥).

Furthermore, numerous endeavors have been made to optimize caching policy, determining what to cache. The selective caching of frequently accessed (*hot*) data stands out as a particularly effective method for improving cache hit rates in environments having uneven access patterns (⑦).

		System	Description
Batching	①	NEVE [44]	Coalesce and defer traps to the hypervisor by logging to avoid context switch overhead between a VM and the hypervisor.
	②	EAW [13]	Exclude outdated data from the batch by allowing modifications in the log before committing.
	③	IX [11]	Apply batching at every stage of the network stack, including system calls and hardware queues to enhance instruction and data locality.
Caching	④	Drizzle [60]	Reuse scheduling decisions across micro-batches as computation is largely static and undergoes infrequent changes.
	⑤	Protean [27]	Cache previous virtual machine allocation result and reuse the placement across multiple requests.
	⑥	Tasi *et al.* [58]	Decouple permission checking from locating a directory and memoizes permission check resuls to enable fast path.
	⑦	NetCache [32]	Let programmable switches to cache hot data for key-value store architectures.
Precomputing	⑧	Duet [9]	Reorder storage maintenance to prioritize data already cached in memory.
	⑨	Itasks [18]	Proactively trigger memory reclaiming upon detecting the first sign of memory pressure to reduce garbage collection during critical path.
	⑩	Correctables [26]	Prefetch dependent objects based on preliminary view instead of waiting for fully consistent one to hide the latency of strong consistency.
Deferring	⑪	Sparrow [51]	Delay assignment of tasks to worker machine until workers are prepared, mitigating race conditions among multiple distributed schedulers.
	⑫	PACTree [35]	With optimistic version lock, perform an operation optimistically without holding a lock and then check the version number.
	⑬	SKYROS [23]	Defer expensive ordering and operation execution until their effects are externalized.
Relaxation	⑭	KnightKing [65]	Allow for sampling the next edge in random walk algorithms without scanning all out-edges at the walker's current vertex.
	⑮	RSS and RSC [28]	Relax certain real-time guarantees of strict serializability for causally unrelated transactions or operations.
	⑯	SAUCR [6]	Relax durability and availability when most servers crash simultaneously to achieve performance comparable to memory-durable one.
Contextualization	⑰	MemLiner [61]	Let application threads notify garbage collection tracing threads about accessed objects for better alignment of memory accesses.
	⑱	Syrup [34]	Use eBPF to have customized scheduling policies defined by applications.
	⑲	CLoF [15]	Automatically generates various hierarchical combinations of locks and selects the most performant configuration.
Hardware	⑳	Nap [62]	Introduce NUMA-aware layer over existing persistent memory (PM) indexes to manage hot items and avoid costly remote PM access.
	㉑	FAERY [66]	Adopt FPGA to accelerate embedding-based retrieval system which requires both high memory bandwidth and pipeline parallelism.
	㉒	KV-Direct [40]	Use programmable NICs to extend RDMA primitives and directly handle key-value operations in the host memory.
Layering	㉓	The nanoPU [29]	Route incoming messages directly to the running threads' registers, bypassing traditional memory hierarchies.
	㉔	TinyNF [55]	Simplify buffer management layers by removing buffer pool. Merge reception, processing and transmission into a single logical flow.
	㉕	RackSched [69]	Two-layer scheduling framework decouples inter-server scheduling in top-of-rack switch and intra-server scheduling in each server.

Table 2: List of example papers adopting each methodology to improve either latency or throughput.

3.3 Precomputing

Precomputing optimizes performance by strategically moving tasks to an earlier time, allowing them to be executed in advance of their actual need. Tasks can be moved to earlier time even before epochs start, as illustrated in Table 1 (d). Although moving a task forward may not directly reduce its execution time, it can optimize the performance of future performance-sensitive executions by moving the task off the critical paths (P_{rm}) and leverages resources that would otherwise be idle or less contended. On the other hand, task can be rearranged within an epoch to seek better opportunity for locality and reduce their execution time (P_{ord}). Duet (⑧) in Table 2 serves as an exemplary case which reorder tasks to better utilize caching.

Executing work earlier relies on two types of speculations. Firstly, it involves speculating that the task will be required in the near future. For example, proactive task execution serves as a preventive measure against unexpected overhead during critical operations, ensuring better control over their execution (⑨). Secondly, precomputing speculatively executes tasks based on current information, rather than waiting until a complete view of information is available (⑩). This proactive approach can significantly enhance system performance by ensuring the tasks are ready when needed, thus minimizing waiting times and accelerating overall workflow. However, to fully leverage the benefits of precomputing, it is crucial to make precise decisions about when and which tasks to execute ahead of time, as incorrect speculation may introduce overhead.

3.4 Deferring

In contrast to precomputing, deferring postpones a task to a later time, employing a lazy approach. Tasks can either be delayed to beyond an epoch (P_{rm}) or shifted to a later time within an epoch (P_{ord}).

One primary motivation for deferring a task is the anticipation of its eventual reduction of runtime when postponed. For instance, deferring a task allows it to make better decisions with the most up-to-date information available (⑪). Furthermore, as previously mentioned, the strategy of deferring tasks can be effectively combined with batching opportunities, since multiple deferred tasks can be executed together. NEVE (①), given as a batching example, also exploits deferring, demonstrating the close relation between these two methodologies (§4.2).

Some deferring examples even anticipate that the task

may eventually be canceled. If there's a possibility that a task could be useless, it can be considered to be deferred. Should it indeed prove unnecessary, the task can then be skipped, thereby preventing the system from wasting resources on tasks that were never required. *Optimistic decision* exemplifies this scenarios, rooted in the assumption that certain events such as concurrency conflicts (⑫) or crash will not occur, thereby deferring their handling until they actually happen.

Another rationale for deferring tasks is to prioritize the performance-sensitive sequence S_n, even if it may compromise later performance. In scenarios where a task within the S_n isn't immediately necessary and can be deferred, it can be executed lazily after S_n has been completed (⑬).

3.5 Relaxation

In some cases, precise or thorough answers are unnecessary while a reasonably accurate approximation suffices. When the workload and context can tolerate a modest degree of inaccuracy or inconsistency, existing system can be modified to reduce task size through approximate execution, thus delivering faster results. As depicted in Table 1 (e), relaxation substitutes existing tasks with significantly shorter tasks (P_{rep}), or even completely skips them (P_{rm}).

For example, sampling, a prevalent method of relaxation, analyzes a small subset of data instead of the entire dataset (⑭). Despite working with a smaller portion of the data, sampling aims to provide insights that reflect the overall dataset, thereby facilitating faster analysis and decision-making. In addition to the accuracy, other software requirements such as consistency (⑮) or availability (⑯) may be compromised in pursuit of performance.

3.6 Contextualization

There is typically a semantic gap between system design and the execution of real workloads. Design decisions often aim to meet general requirements due to uncertainty about future usage, but this lack of workload awareness can hinder optimal performance. To bridge this gap, existing systems are often customized by incorporating runtime context into their decision process. This context is often obtained by collecting data available during execution (⑰). Table 1 (f) illustrates a case where a system performs additional tasks to inform future decisions. While these may incur runtime overhead (denoted as x_7), the resulting performance (P_{rep}) improvements generally justify the cost. Alternatively, profiling workload characteristics off the critical path enables specialization without runtime penalties (⑱). eBPF also helps bridge the semantic gap by letting applications guide the kernel toward more efficient, workload-aware behavior (⑲).

3.7 Hardware specialization

Customizing the general system design for specific hardware configurations is another common approach. Optimization for hardware primarily involves determining which tasks should be executed on which hardware, taking into account the workload characteristics and hardware specifications. For example, numerous studies tailor existing general systems for specific hardware requirements such as non-volatile memory, non-uniform memory access (⑳), FPGA (㉑), or SmartNIC (㉒). In Table 1 (g), task x_4 is executed on different hardware (F') which can execute it more efficiently, resulting in reduced runtime ($F(x_4) > F'(x_4)$). When optimizing an existing system for particular hardware, the advantage of utilizing the hardware must outweigh the software overhead (x_7, x_8) incurred in transitioning to that hardware.

3.8 Layering

A layer is a logical division or abstraction defined during system design to organize and structure a system's components. Each layer encapsulates a specific set of functionalities or services, helping to modularize the system and making it easier to understand, maintain, and scale. While layers provide clear structural benefits, they also introduce performance overhead, stemming from additional processing, communication, or indirection across layers.

Layering, the final methodology, optimizes sequential performance by reducing the overhead associated with layers—through merging multiple layers, decomposing layers more efficiently, or selectively bypassing layers where appropriate.

Bypassing. It involves creating shortcuts or alternative paths in a system to avoid unnecessary steps or processes, thus reducing latency and improving overall performance. In Table 1 (h), x_4 is completely skipped through a fast-path (P_{rm}). Many existing works bypass the kernel stack [5, 16, 19, 21, 29, 37, 40, 46, 48, 54, 57, 68] and handle operations within the user space to eliminate context switching overhead between user and kernel, thereby having more flexible control for runtime requirements (㉓).

Delayering. It reduces the number of layers in a system's architecture. This simplification consolidates multiple abstraction layers and removes software overhead between them (㉔). Delayering often sacrifices generality, flexibility or functionality to achieve improved performance in specific scenario. Table 1 (h) shows a scenario where two tasks are merged into a single, larger layer, effectively reducing overall runtime. Delayering not only combines existing layers into one but also minimizes the intermediate overhead, leveraging both P_{rm} and P_{rep}.

While both delayering and bypassing aim to reduce the number of layers being executed, their approaches differ. Delayering restructures the system architecture itself to have fewer layers, thereby simplifying the system. In contrast, by-

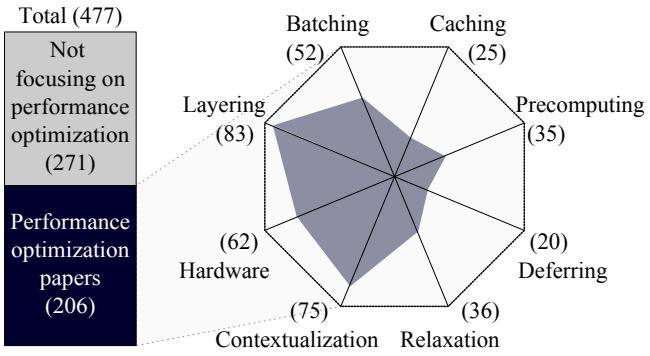


Figure 2: Number of papers leveraging each methodology. Since each paper may employ multiple methodologies, the cumulative count of methodologies exceeds the total number of papers within our scope.

passing selectively skips over existing layers without modifying the underlying architecture—the skipped layers remain present and may still be executed in other contexts.

Decoupling. In contrast to delayering, decoupling splits one layer into multiple ones. Decoupling aims to minimize dependencies between different components or modules within a system. This strategy enhances finer control and execution by loosening the coupling between parts, allowing changes to be made to one component without affecting others. Furthermore, by loosening dependencies between layers, it opens more opportunity for parallelization. In Table 1 (h), x_2 is decoupled into x_2^1 and x_2^2. Despite the potential for additional overhead between the layers due to having more layers, the benefit arises from the finer control it provides. For example, it becomes possible to execute either x_2^1 or x_2^2 individually rather than executing the larger x_2, or to implement different logic for each task (㉕).

4 Case Study

4.1 Statistics

To empirically evaluate the completeness of the eight methodologies in enhancing system performance, we systematically analyzed entire papers from the OSDI and SOSP conferences over the last decade, from 2013 to 2022. The review process was rigorously conducted, with each paper independently reviewed by two different reviewers to ensure accuracy and minimize bias. Papers from 2023 and 2024 were intentionally excluded from this analysis to serve as a fresh dataset for evaluating our fine-tuned model, as discussed in Section §5.

As depicted in Figure 2, our analysis of 477 papers reveals that 271 do not focus on performance optimization of sequential tasks. Among the remaining 206 performance-related papers, every technique employed to enhance existing performance aligns with one of our eight methodologies. Furthermore, these methodologies are often used in tandem. On average, research papers utilize approximately 2.01 methodologies, demonstrating the necessity of a multidimentional approach for improving system performance.

Table 2 provides a detailed breakdown of file and storage system papers presented at SOSP 2021, serving a case study to illustrate the practical application of the methodologies within the given domain. It provides insights into how the eight methodologies are leveraged in practice to address challenges within file and storage systems. Additionally, we provide suggestions for further performance optimization directly derived from the methodologies.

4.2 Relations between methodologies

The optimization of system performance through the eight methodologies is not isolated; rather, these strategies often interrelate and amplify one another when applied together.

Batching and deferring are closely interwined optimization strategies. Batching aggregates multiple operations to execute them collectively, inherently modifying their execution timing. For instance, a write buffer exemplifies the combined use of batching and deferring, as it accumulates multiple write operations to execute them together at a later time, improving data transfer and write efficiency. This approach not only minimizes repetitive, immediate operations but also strategically defers execution to a more opportune moment, thereby leveraging both batching and deferring for enhanced performance.

Decoupling separates tightly integrated components into finer-grained subtasks, enhancing modularity and flexibility. This separation facilitates a more targeted optimization of individual components. Often, specialization tailored for specific hardware configurations or software requirements necessitates initial decoupling of integrated mechanisms. This ensures that modifications are minimal and do not inadvertently impact any other system components. For example, Akkio [10] introduces an intermediary layer between client applications and the datastore systems to separate the allocation of shard data from its management. Consequently, this allows application to specify their own shard sizes much smaller than existing one, and determines the distribution of data across shards, facilitating customized shard assignment tailored to the needs of each application. Similarly, decoupling can enable more effective caching opportunities, especially when only a subset of a task's output is reused over time. For instance, Tsai *et al.* [58] first decouple permission checking from directory lookup, allowing permission results to be cached and reused independently, creating a fast path for common operations.

4.3 Applying methodologies to real problem

In exploring practical applications of the methodologies to improve system performance, we bring one case study for

Paper	Methodologies used in the paper	Suggestions for further optimization
ccNVMe [43]	❶ Instead of having per-request communication, batch requests and ring doorbell per transaction to reduce MMIOs and better utilize the write combining buffer. ❼ Leverage data dissemination mechanisms of NVMe for crash consistency.	❻ Instead of always chaining the completion doorbell to order transactions, enforce it only for dependent transactions.
Kangaroo [49]	❶ Batch multiple objects in log-structured cache first and move objects to set-associative cache as a group to reduce write amplification.	❽ Objects are always written twice: once in KLog and once in KSet. Having additional fast-path to bypass KLog and write directly to KSet could be beneficial in certain scenarios.
IODA [41]	❸ Proactively reclaim enough space via garbage collection to guarantee it is not triggered during predictable window. ❻ To reduce semantic gap between host and SSD for predictable latency, add new fields to I/O submission command and quickly get contextual status.	❷ Introduce a caching layer to serve hot items while the underlying flash storage is busy.
HeMem [56]	❶ Sampled memory events are processed in batches. ❶ Migrate memory pages in batches. ❺❻❼ Monitor applications' memory usage by sampling and make memory management decisions based on these patterns and hardware characteristics.	❸ Based on the memory access pattern, speculatively prefetch memory page from NVM to DRAM to lower the access latency.
J-NVM [39]	❽ Decouple object into proxy and data to directly access NVMM without runtime garbage collection. ❽ Remove the cost of converting data back and forth between NVMM and Java representation.	❷ Since persistent objects are separated into persistent data and a proxy in J-NVM, cache only the persistent data to hide access latency to NVMM.
PACTree [35]	❸ A reader optimistically performs a read on a data node. ❽ Decouple search layer and data layer and asynchronously execute structural modification operations. ❼ Take into account the specific characteristics of Non-Volatile Memory (NVM) to design an efficient persistent index.	❶ Given that allocating persistent memory incurs significant costs, preallocate memory in batches to hide allocation latency in critical path.
Aurora [59]	❽ Instead of having volatile memory and persistent storage separately, transparently persists all application states at regular intervals.	❶ Aurora can leverage batching to persist relevant POSIX objects together and reduce the latency of persistence.
WineFS [33]	❶ More huge pages to have less number of TLB entries ❸ Proactively take actions to prevent future fragmentation. ❻ Collects more runtime information for better allocation.	❷ Instead of traversing red-black trees repeatedly to find unaligned free space, cache the results of the traversals for faster allocation of small memory requests.
uFS [46]	❽ The uFS servers are implemented on userspace and bypass existing operating system stack.	❸ Preemptively reads more data than requested when the data access patterns are predictable.
FragPicker [52]	❶ Rather than issuing multiple small I/O requests due to fragmentation, consolidate them by migrating data to contiguous storage locations. ❻ Monitor applications' I/O activities and selectively migrate data for defragmentation.	❷ To efficiently address defragmentation, FragPicker should operate periodically. Caching previously observed I/O patterns might expedite the analysis phase in subsequent executions, for instance, predict which data blocks are more prone to fragmentation.
Witcher [22]	Propose automatic testing framework to detect correctness and performance bugs in NVM-based persistent key-value stores.	The goal of the paper is out of our scope.

Table 3: A complete list of file and storage system papers presented at SOSP'21. Each paper is annotated with the corresponding methodologies employed to improve performance: (1) batching, (2) caching, (3) precomputing, (4) deferring, (5) relaxation, (6) contextualization, (7) hardware, and (8) layering. Suggestions for further optimizations, derived from these methodologies, are also provided.

a paper, *Application-informed kernel synchronization primitives* [53] from OSDI'22. The paper introduces SynCord, a framework enabling application developers to deploy custom behaviors for kernel locks. Kernel locks are synchronization primitives within the kernel, to ensure that concurrent threads do not simultaneously access shared resources or critical sections of code. When multiple threads simultaneously request the same lock to enter a critical section, the lock implementation internally organizes these threads into a waiting queue. This process ensures orderly lock acquisition and allows only one thread to enter the critical section at any moment. Consequently, the critical section and the lock implementation are fundamentally serial (F_{serial}) and cannot benefit from parallelization. Optimizing these serial components, therefore, holds the potential to significantly improve performance.

Figure 3 demonstrates the strategies employed by SynCord to enhance lock performance and how they align with our methodologies. Primarily, SynCord maintains concurrent threads in a waiting queue and allows dynamic rearrangement of the waiters. A key contribution of SynCord is its decoupling of the reordering policy from lock mechanism, allowing the policy part to be managed in userspace (❶). Consequently, it allows userspace to create its own rule for lock acquiring order, facilitating the specialization of kernel lock behaviors to meet specific workload demands or requirements (❷). With user-defined policy, SynCord dynamically reorders threads in a waiting queue instead of conventional

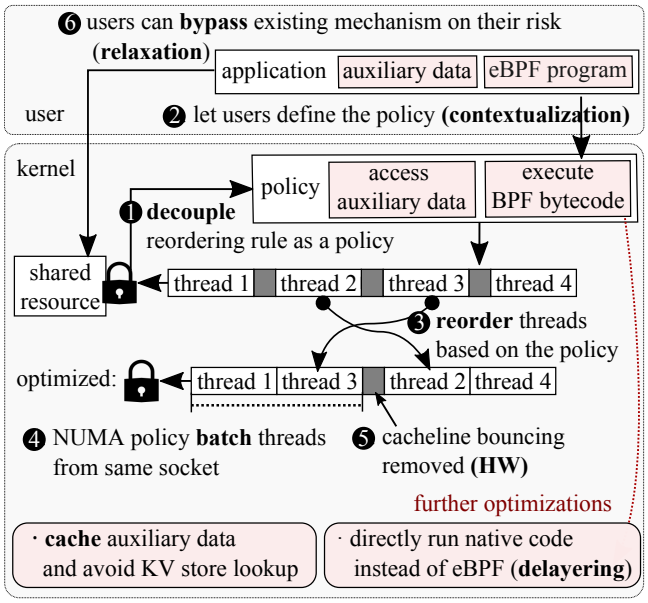


Figure 3: SynCord [53]'s strategies with integration of our methodologies. SynCord leverages most of the methodologies to optimize the performance of kernel locks. Moreover, using the methodologies, we identified two potential optimization opportunities: caching and delayering.

first-in-first-out policy (❸). For instance, a thread may be prioritized and moved to the front if it has a short critical section, or it can be deferred to the back if it operates on a slower core. This strategic reordering is carried out with the underlyig hardware in mind, as illustrated by batching threads from the same NUMA node together (❹) to remove cacheline bouncing between them (❺). Moreover, SynCord provides an API to completely bypass the kernel lock, allowing it to be redefined in userspace at the user's own risk (❻). While this approach relaxes the safety guarantee, it unveils opportunities for performance enhancements.

While SynCord incorporates the majority of our methodologies, two methodologies are still not directly adopted in their work, which hold potential for further optimization. The first one is caching. To accommodate various user-defined lock policies, SynCord provides auxiliary data through shadow variables [45], stored as a key-value store in the kernel. In scenarios where lock performance is crucial, it is common for threads to repeatedly acquire and release locks aggressively. Therefore, introducing a caching mechanism for this auxiliary data would allow for the reuse of previously accessed data upon subsequent accesses, thereby further reducing the overhead associated with repeatedly searching the key-value store every time using a lock.

Another overlooked opportunity is delayering. To protect the kernel from potentially faulty user code, SynCord compiles user-defined policies into eBPF [20] bytecode, allowing for static analysis through the eBPF verifier. However, the execution of eBPF program introduces additional overhead with each use of a lock. One potential optimization involves compiling user-defined policy programs into native code once they pass the static verifier, enabling their direct execution. This approach would eliminate the intermediary overhead between lock mechanism and user-provided policies.

In summary, this case study illustrates a successful application of methodologies. Moreover, exploring additional methodologies such as caching and delayering presents potential avenues for further optimization. It not only demonstrates that these methodologies are applicable in real research works, but also highlights its role as a checklist to maximize performance optimization and minimize any potential opportunities to be missed.

5 SysGPT: Hints for System Performance

Devising effective solutions for system performance optimization remains an open-ended and intuition-driven, often relying on experience rather than structured guidance. While tools for identifying bottlenecks and evaluating outcomes are mature, the intermediate step—formulating an actionable solution—often lacks systematic support. To address this gap, we introduce SysGPT, a fine-tuned [4] large language model (LLM) that provides context-aware optimization suggestions.

Our goal is to make our methodologies more applicable to real-world optimization workflows. While standard LLMs like GPT-4 [3] are trained on broad corpora, they often produce vague or overly verbose outputs that lack the specificity required for system-level decisions. We thus investigate whether a model that internalizes the distilled insights from a decade of systems research can provide more targeted and practically feasible suggestions.

5.1 Implementation

To build SysGPT, we first constructed a fine-tuning dataset based on our comprehensive review of all SOSP and OSDI papers published from 2013 to 2022. This corpus contains a wide range of system performance problems, each annotated with a problem description, observations, and the solution strategy employed in the original paper. The solution is annotated with both the applicable methodology from our taxonomy defined in §3, and a concise explanation of how it was used to address the problem. The dataset spans diverse systems domains, including file and storage systems, databases, machine learning infrastructure, and more. Notably, we excluded papers from 2023 and 2024 to use them exclusively for evaluation purposes, as described in §5.2 and §5.3, ensuring that SysGPT is tested on unseen examples.

Using this dataset, we fine-tuned GPT-4o [2] to produce SysGPT, a domain-specialized LLM that provides structured suggestions for optimizing sequential system performance.

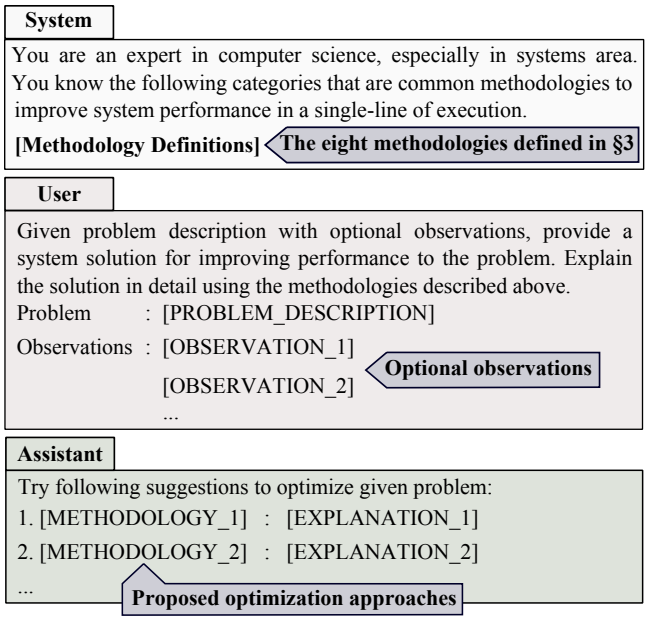


Figure 4: The prompt used for fine-tuning SysGPT. Observations can include background, existing challenges, related work, and more. Based on the descriptions provided by *system* and *user*, the *assistant* proposes multiple suggestions for optimizing system performance. Each suggestion is explicitly tied to the eight methodologies, accompanied by a brief explanation of its application.

The prompt format used during training is illustrated in Figure 4. Once fine-tuned, SysGPT can be used interactively. Given a system performance problem and context, the model returns a set of optimization strategies, each rooted in our methodology and supported by brief justifications.

To support reproducibility and facilitate future research at the intersection of systems and LLMs, we release our complete dataset and evaluation benchmark [1].

5.2 Qualitative results

We first examine whether SysGPT produces more feasible and meaningful suggestions than baseline models. Evaluating the *quality* of optimization suggestions is inherently challenging. There is currently no well-established benchmark for this task, and much of the utility of a suggestion depends on how it is interpreted and applied in practice. Since the effectiveness of a response often becomes clear only when a user reads and reasons about the suggestion in context, we include in this section the actual outputs generated by SysGPT and baseline GPT-4 for direct inspection and comparison.

Table 4 compares responses from SysGPT and baseline GPT-4. The test set consists of all database-related papers from SOSP and OSDI in 2023, which were excluded from our training data. To ensure comprehensive coverage and

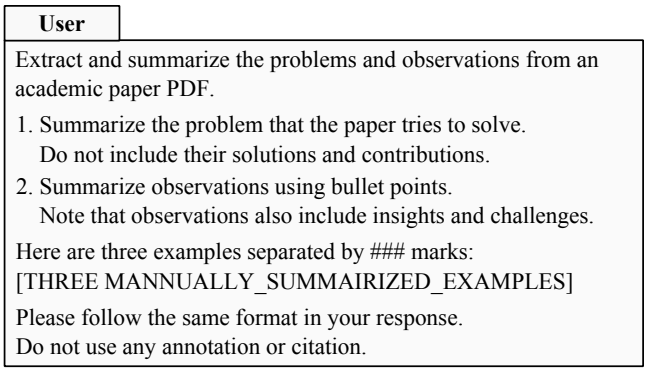


Figure 5: The prompt to automatically extract *problem descriptions* and *observations* from uploaded papers. The prompt explicitly instruct the model to identify and extract only the relevant problems and observations, while excluding any proposed solutions.

eliminate selection bias, we include the complete list of database papers, even those that do not primarily focus on performance optimization. Among the 10 papers, six address performance-related problems, while the remaining ones focus on topics such as security, privacy, or maintainability. To ensure scientific rigor, we use only the first model output for each case, with the temperature set to 0 to ensure deterministic behavior. Due to space constraints, the table contains summarized sentences, although most are directly extracted from the full model outputs.

To eliminate human bias in preparing inputs for test set, we automatically extracted problem statements and observations from each paper using a separate GPT-4o model with knowledge retrieval capabilities. The extraction prompt shown in Figure 5 explicitly instructs the model to include only the problem and relevant observations, while omitting any solutions or contributions described in the paper. This setup ensures that both models are evaluated solely based on their own reasoning given the same input, and that the evaluation pipeline is fully automated without any human intervention.

Since the problems and observations are directly derived from the original papers, a strong model should generate suggestions that meaningfully align with the solutions proposed by the authors. In this context, SysGPT consistently produces outputs that exhibit a high degree of correspondence with the ground-truth solutions, often matching not only the methodology but also specific implementation details.

Examples include concrete recommendations such as "set different lifetimes for hot/cold data by storing them separately in cache" or "decouple range index into frontend staging buffers and backend index update". In contrast, GPT-4 tends to produce more generic responses, such as "leverage runtime heuristics" or "implement an in-memory caching layer", which are less specific and often lack actionable guidance.

DeToX [12]	**P+O**	Traditional caching policies focus on object hit rate, but transactional workloads need to cache all concurrently requested objects to reduce latency. Transactions modeled as DAGs require scoring for eviction, which is costly, infeasible, and complicated by overlapping object usage.
	GPT-4	Design a caching policy that prioritizes key groups in transactional DAGs, leveraging runtime heuristics to balance latency and computational complexity.
	SysGPT	Infer transactional DAGs using dynamic profiling to capture transaction locality (❻). Pre-fill the cache with keys (❸) based on dependency tracking.
	Solution	To limit the number of groups scored, keys interchangeable in reducing critical length are treated as one group. Dynamically infer groups when DAGs are inaccessible (❻). Preemptively cache related items based on dependency tracking (❸).
VBASE [67]	**P+O**	Relational databases rely on strict monotonicity, while vector systems use approximate top-K, lacking this property. This division forces hybrid queries, like finding top-K vectors meeting relational conditions, to handle similarity search and relational filtering separately, resulting in inefficiencies.
	GPT-4	Dynamically adjust indices based on query patterns and embedding techniques. Employ adaptive sampling strategies to balance precision and computation. Integrate scalar and vector constraints to minimize computational overhead.
	SysGPT	Integrate vector search with relational databases by developing a query engine tailored for hybrid workloads (❽). Cache tentative results to avoid redundant computation (❷) while incrementally exploring new data. Use an adaptive search model that allows early termination based on the size of the result set during the search (❻).
	Solution	Instead of having two separate phases—one for finding top-K vectors and another for applying relational query filters—unify these steps into a single execution engine (❽) that incrementally expands traversal and terminates early once the query conditions are met. Use runtime information to traverse high-quality indices more frequently (❻).
Falcon [30]	**P+O**	The introduction of persistent cache challenges existing designs of online transaction processing engines on non-volatile memory. Cache-NVM granularity mismatch causes write amplification, and out-of-place updates increase NVM writes and require frequent index updates.
	GPT-4	Eliminate traditional logging by ensuring atomic updates directly in persistent cache. Employ fine-grained in-place updates to minimize NVM writes and integrate cache-aware index structures.
	SysGPT	Eliminate explicit cache line flushing by introducing persistent cache (❼). Set different lifetimes for hot/cold data by stroing them separately in cache (❷). Align cache eviction policies with NVM write block sizes (❶).
	Solution	With persistent cache, no need to explicitly flush logs for persistency (❼). To reduce write amplification from granularity mismatch, multiple cache lines are merged into one NVM write (❶). A small LRU cache absorbs hot tuples, avoiding manual flushing and reducing NVM writes (❷).
SPFresh [64]	**P+O**	Updating high-dimensional vector indices is challenging due to the cost of maintaining index quality during frequent updates. Existing methods rely on costly periodic global rebuilds, result in fluctuating query latency and accuracy.
	GPT-4	Adopt a hybrid approach combining cluster-based local updates with lightweight graph pruning. Cluster-based indexing minimizes the cost of local modifications, while periodic, asynchronous graph pruning ensures global quality without frequent full rebuilds.
	SysGPT	Incrementally update graph indices with minimal distortion on global indices (❺). Only manage short range connections accurately at runtime (❻) and move partition maintenance tasks off the critical search path (❸). Batch updates locally (❶).
	Solution	Perform incremental updates by reassessing only vectors affected by local changes, rather than rebalancing the entire index (❺). Accumulate small updates locally in partitions (❶) and use existing centroids and data distribution to expedite rebalancing (❷). Move the background split-reassign off from the critical path of update (❸).
Chardonnay [17]	**P+O**	Existing systems rely on two-phase commit for atomicity, causing performance limitations due to high RPC and log flushing latencies. Modern hardware with μs-level latency alleviates these issues, shifting the bottleneck to disk read. This creates contention during lock-holding periods, particularly under high workloads.
	GPT-4	Implement an in-memory caching layer for frequently accessed data. Leverage low-latency storage and fast RPCs to asynchronously synchronize disk-based storage.
	SysGPT	Relax synchronization for reads and eliminate dependencies between transactions (❻). Preemptively aggregate information about transaction participants and prefetch data (❸). Group read and write operations requesting same locks (❶, ❻).
	Solution	Use fast RPCs to read a counter for strict serializability without locks (❼). Dry run transactions to approximate read sets and prefetch them (❸). Reorder lock acquisition order based on the precomputed read/write sets to avoid deadlocks (❻).
ScaleDB [50]	**P+O**	Scaling in-memory databases on multi-core architectures is hindered by contention on range-index structures like B+ trees, caused by synchronous updates for consistency. However, range scans typically operate on stale data and read a few records.
	GPT-4	Decouple updates and reads by leveraging append-only logs for write operations. Large read-only range scans can bypass the index to directly access the underlying data.
	SysGPT	Decouple range index into staging buffers for front-end and backend index update (❽). Apply index updates in batches while caching them in the staging buffers (❶, ❷). Delay updating index from the critical path of transaction processing (❹).
	Solution	Decouple transaction execution from updating the affected range index (❽). Asynchronously apply batched updates to range indexes with optimistic concurrency control (❶, ❹). Cache recently committed transactions separately (❷).
Trubo [36]	**P+O**	The goal of the paper is user privacy, which is out of our scope.
TxCheck [31]	**P+O**	The goal of the paper is finding transactional bugs, which is out of our scope.
K9db [7]	**P+O**	The paper focuses on tracking data ownership in database and easier maintenance based on it.
HEDB [42]	**P+O**	The goal of the paper is security and maintainability, which are out of our scope.

Table 4: A complete list of database papers from SOSP'23 and OSDI'23, showing the problem and observations (P+O), suggestions from standard GPT-4 and SysGPT, and the actual solution. Solutions and suggestions are tagged with corresponding methodologies: (1) batching, (2) caching, (3) precomputing, (4) deferring, (5) relaxation, (6) contextualization, (7) hardware, (8) layering.

User

Evaluate two answers to a system performance problem by comparing them to the standard solution. Your goal is to determine which answer aligns more closely with the standard solution and explain your reasoning.

Problem statement : `[problem description]`

Observations : `[observations]`

Standard solution : `[Actual solutions used in the paper]`

Answer to evaluate : `Answer 1 = [Answer from GPT-4]`
`Answer 2 = [Answer from SysGPT]`

Figure 6: Prompt. LLM-based comparative evaluation.

To further validate this distinction, we conducted an LLM-based comparative analysis following standard practices in language model evaluation. In this setting, GPT-4o [2] serves as an independent evaluator, tasked with judging which of the two model outputs more closely aligns with the actual solution from the paper. The prompt for this evaluation is shown in Figure 6.

For 37 out of 42 papers (88%) in the test set (§5.3.2), the evaluator preferred SysGPT's output over that of the baseline. This strong preference stems from SysGPT's ability to provide more relevant and detailed suggestions that better align with the actual solutions devised by systems researchers. For example, in the case of Nomad [63], the evaluator noted that SysGPT's response "directly addresses two of the three key strategies: contextualization and deferring, with a clear focus on optimizing memory management based on application behavior and reducing unnecessary migrations." This demonstrates how SysGPT's methodologies enable it to generate more targeted and actionable suggestions for real system problems.

For the remaining 5 papers where GPT-4o's output was preferred, the evaluator's rationale primarily focused on broader strategy coverage rather than suggestion quality. For instance, in the case of CHIME [47], the evaluator noted that while GPT-4o's response covered all three components (batching, contextualization, and precomputing), it lacked critical implementation details like RDMA usage and hotspot buffer design. In contrast, SysGPT provided more practical and detailed suggestions for batching and precomputing, though it missed the contextualization aspect. This highlights SysGPT's strength in delivering concrete, actionable recommendations, even if it occasionally prioritizes depth over breadth in its coverage of optimization strategies.

5.3 Quantitative results

While the qualitative results in §5.2 highlight the specificity and feasibility of model-generated suggestions, it is inherently interpretive. To provide more objective evidence, we also include a quantitative evaluation that assesses how accurately models can identify applicable optimization methodologies given a system performance problem.

5.3.1 Task definition

We frame the evaluation as a multi-label classification task: given a system performance problem and its associated observations, the model must predict a set of applicable methodologies from the eight defined in §3. We then evaluate how well these predictions align with the methodologies actually used in the corresponding paper.

This formulation enables measurable evaluation using standard classification metrics such as precision, recall, and F1-score. It also helps reduce the subjectivity involved in open-ended solution generation by focusing on whether the model correctly identifies the underlying optimization principles.

5.3.2 Evaluation setup

The test set consists of all performance-related papers published at OSDI and SOSP in 2024 (42 out of 96). These papers were not only excluded from the fine-tuning dataset, but also released after the knowledge cutoff date (September 30, 2023) of both the base GPT-4o model [2] and SysGPT. As a result, neither model has access to the content of these papers, ensuring a fair evaluation on truly unseen examples.

Each paper is annotated with ground-truth methodologies based on the actual solution described by the authors. We compare three approaches: (1) the baseline GPT-4o model, (2) GPT-4o with few-shot prompting using 3, 5, and 10 examples, and (3) our fine-tuned SysGPT.

Few-shot examples are drawn from our training set and are carefully selected to cover all eight methodologies when 10 examples are given ($N = 10$). Each prompt builds on the structure shown in Figure 4. It includes a brief description of the taxonomy in the system prompt, along with N representative input-output examples and formatting instructions for the model's output.

All models are evaluated using the same inputs—problem descriptions and observations—automatically extracted as described in Figure 5. This ensures that any performance differences reflect the models' reasoning capabilities rather than differences in prompt quality, output format, or prior exposure to the test cases.

Metrics. We report micro-averaged precision (the fraction of predicted methodologies that are correct), recall (the fraction of ground-truth methodologies that are correctly identified), and F1-score (the harmonic mean of the precision and recall) over the eight methodology labels. Partial matches are accounted for; for example, if the ground truth is (batching, caching) and the model predicts (batching, precomputing), this counts as one true positive, one false negative, and one false positive.

Model	N	Output	Precision	Recall	F1-score
GPT-4o	0	-	0.277	0.934	0.426
GPT-4o	0	Top-2	0.476	0.377	0.421
GPT-4o	3	-	0.342	0.840	0.486
GPT-4o	3	Top-2	0.560	0.443	0.495
GPT-4o	5	-	0.344	0.802	0.482
GPT-4o	5	Top-2	0.547	0.434	0.484
GPT-4o	10	-	0.345	0.868	0.479
GPT-4o	10	Top-2	0.536	0.425	0.474
SysGPT	-	-	**0.758**	**0.651**	**0.701**

Table 5: Methodology prediction evaluation on the SOSP/OSDI'24 test set. We compare SysGPT and GPT-4o under few-shot prompting ($N = 0, 3, 5, 10$) with/without top-2 output constraint.

5.3.3 Results

Table 5 summarizes the results of our evaluation. SysGPT clearly outperforms all GPT-4o configurations in F1-score, demonstrating that fine-tuning with methodology-aware supervision leads to more accurate and relevant predictions. In its default form, GPT-4o tends to enumerate nearly all methodologies regardless of input context, resulting in an average of 6.23 suggestions per query. This behavior leads to a high number of false positives—resulting in low precision and high recall—and ultimately producing noisy and unhelpful outputs.

To compensate for GPT-4o's tendency to overgenerate, we added a constraint limiting its output to the top-2 most promising methodologies. While this adjustment improves GPT-4o's precision, now the recall drops significantly, and the overall F1-score remains lower than that of SysGPT. We believe this improvement stems from SysGPT's exposure during fine-tuning to realistic solution patterns with properly labeled methodologies. This training enables SysGPT to more effectively filter out irrelevant suggestions and focus on feasible strategies grounded in the problem and observations.

To assess robustness under non-deterministic sampling, we evaluate performance (Best@k) across multiple temperature settings ($T = 0.1, 0.3, 0.5, 0.7$) and trials ($k = 1, 3, 5, 10$), reflecting what an average user might expect from multiple trials. For each temperature, we generate k samples and report the best-scoring prediction relative to ground truth. As shown in Figure 7, SysGPT consistently achieves higher scores across all temperature settings and trial counts, with an average F1-score improvement of 39.1% over GPT-4o. This demonstrates that SysGPT's methodology-aware training enables more stable and reliable predictions even under varying sampling conditions.

These results validate that SysGPT's performance gains are not simply due to stylistic alignment with the training data, but stem from its improved capability to identify relevant optimization principles. The methodology-aware fine-tuning improves both precision (fewer irrelevant suggestions) and recall (more comprehensive coverage), and generalizes to unseen papers from a later publication year. This demonstrates that SysGPT is not merely reproducing past solutions, but learning transferable reasoning patterns applicable to new system performance problems.

6 Discussion

6.1 Complementing causal profiling

Performance optimization is a multifaceted challenge. Optimizing sequential code is not a panacea, and improvements on one task do not always translate to end-to-end performance gains. As demonstrated by Coz [14] paper, accelerating a component may have little or no effect on overall performance if it is not on the critical path. Coz introduces causal profiling, a technique that quantifies the potential impact of speeding up different parts of a program to identify *where* optimizations will be most effective. Once such regions are identified, however, the challenge becomes *how* to optimize them effectively. This is where our work contributes. While Coz identifies high-impact optimization targets, our methodology provides a systematic framework for realizing those optimizations. Together, they form a complementary performance optimization pipeline guiding both where to focus and how to proceed.

6.2 Coordination between tasks

While our framework models performance in terms of individual sequences, coordinating multiple tasks is often crucial for achieving system-wide performance improvements. In multi-threaded and distributed environments, interactions between tasks—such as contention for shared resources, scheduling delays, or communication overhead—can dominate execution time, even when individual operations are optimized. Strategies that align task execution to reduce contention, schedule work based on hardware topology, or overlap communication and computation are widely-recognized techniques for improving throughput and latency.

These coordination strategies, however, typically require reasoning across multiple execution contexts, extending beyond the assumptions of our sequential model. Although some coordination effects—such as improved locality or deferred execution—can be partially captured through batching or reordering, a full treatment of inter-task dynamics remains beyond the current scope. Extending our methodology to incorporate coordination-aware techniques represents a promising direction for future work.

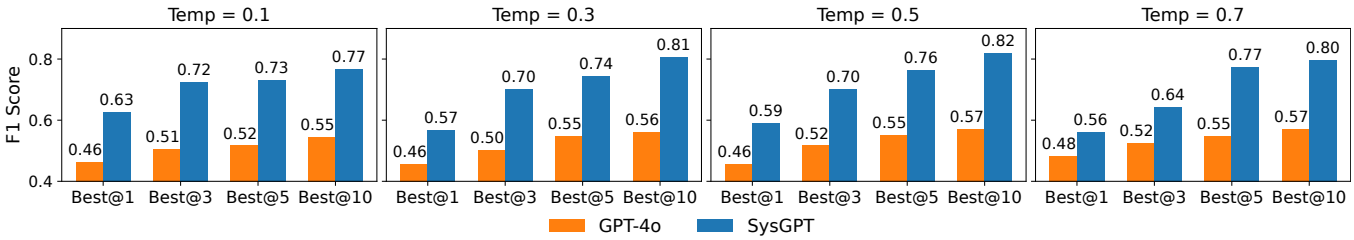


Figure 7: F1-scores under non-deterministic sampling with temperature values set to 0.1, 0.3, 0.5 and 0.7. For each temperature, we generate k responses and report the best score among the k trials (Best@k).

6.3 Limitations of SysGPT

SysGPT targets the second step of performance optimization: devising solutions. Its effectiveness therefore depends on input quality and the subsequent implementation. Well-defined problems and detailed observations yield precise, actionable suggestions, whereas vague inputs can lead to generic ones. SysGPT offers conceptual guidance in natural language but does not directly modify code. As such, implementation and evaluation still require developer expertise.

We envision a future where the entire optimization pipeline—from performance monitoring and solution design to code generation and evaluation—is automated. Realizing this vision will require significant advances across program analysis, synthesis, and systems tooling. Our work takes a step toward this goal, providing a foundation for automated performance optimization.

7 Related Work

Optimizing system performance has been a longstanding aspiration of computer systems research. Alongside countless studies aimed at enhancing performance across various systems through diverse strategies, there have also been some efforts to explore high-level methodologies for performance optimization.

Lampson [38] provides hints and principles for computer system design, suggesting principles such as simplicity, timeliness, efficiency, adaptability, dependability and desirability. While Lampson's discussion of *efficiency* partially aligns with our goal of system performance optimization, it largely relies on intuition and high-level ideas. In contrast, our work takes a systematic approach to optimize system performance in terms of sequential execution, by defining fundamental principles and empirically validating the completeness of the four methodologies with scrutiny over a decade. We illustrate how these methodologies can be practically applied to solve real-world problems efficiently, with case study and SysGPT, significantly extending the existing literature on performance optimization.

The Systems Performance book [25] also provides methodologies to improve performance and reduce latency. However, it targets system administrators and explains practical tools for better system monitoring. Based on the introduction of various performance analysis tools and background to understand them, the book guides how to use existing system and tools better, rather than fundamentally optimizing system designs itself, as our work does.

Similar to the CAP theorem [24], which states that a distributed system can optimize only two of consistency, availability, and partition tolerance, our methodologies define the boundaries of system performance optimization. Both frameworks help narrow the scope of exploration, guiding developers toward feasible and practical improvements.

8 Conclusion

Optimizing sequential system performance remains a fundamental yet often understructured task in systems research. We proposed a systemized framework based on three core principles—removal, replacement, and reordering—and distilled them into eight actionable methodologies commonly used across a decade of systems literature. This taxonomy clarifies the space of optimization opportunities and offers a systematic language for reasoning about performance. Built on this foundation, we introduced SysGPT, a fine-tuned LLM trained to suggest strategies grounded in this methodology. Evaluated on unseen papers, SysGPT significantly outperformed GPT-4 baselines in both qualitative and quantitative evaluations, demonstrating its superior ability to generate targeted, actionable suggestions for real-world system performance problems.

9 Acknowledgment

We thank our shepherd and the anonymous reviewers for their valuable feedback. We are also grateful to Changwoo Min and Sanidhya Kashyap for insightful discussions during the early stages of this work. This research was supported by ONR under grant N00014-23-1-2095, DARPA V-SPELLS N66001-21-C-4024, and gifts from Facebook, Mozilla, Intel, VMware, and Google. This work was also supported by IITP grant funded by the Korean government Ministry of Science and ICT (MSIT) under the project titled RISC-V based system software development for open ecosystem of SDR (No.RS-2024-00459774).

References

[1] Sysgpt. https://github.com/sslab-gatech/SysGPT, 2025.

[2] OpenAI Models: GPT-4o. https://platform.openai.com/docs/models/gpt-4o, accessed April, 2025.

[3] OpenAI Models: GPT-4. https://platform.openai.com/docs/models/gpt-4, accessed December, 2024.

[4] Fine-tuning. https://platform.openai.com/docs/guides/fine-tuning#preparing-your-dataset, accessed December, 2024.

[5] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 353–369, New York, NY, USA, Oct. 2019. Association for Computing Machinery. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359656.

[6] R. Alagappan, A. Ganesan, J. Liu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. {Fault-Tolerance}, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 390–408, 2018. ISBN 978-1-939133-08-3.

[7] K. D. Albab, I. Sharma, J. Adam, B. Kilimnik, A. Jeyaraj, R. Paul, A. Agvanian, L. Spiegelberg, and M. Schwarzkopf. K9db: Privacy-Compliant storage for web applications by construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 99–116, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/albab.

[8] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[9] G. Amvrosiadis, A. D. Brown, and A. Goel. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 457–473, New York, NY, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815424.

[10] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, 2018. ISBN 978-1-939133-08-3.

[11] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. {IX}: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014. ISBN 978-1-931971-16-4.

[12] A. Cheng, D. Chu, T. Li, J. Chan, N. Crooks, J. M. Hellerstein, I. Stoica, and X. Yu. Take out the TraChe: Maximizing (tra)nsactional ca(che) hit rate. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 419–439, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/cheng.

[13] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, Nov. 2013. Association for Computing Machinery. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522724.

[14] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[15] R. L. de Lima Chehab, A. Paolillo, D. Behrens, M. Fu, H. Härtig, and H. Chen. Clof: A compositional lock framework for multi-level numa systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 851–865, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483557. URL https://doi.org/10.1145/3477132.3483557.

[16] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, Oct. 2019. Association for Computing Machinery. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359637.

[17] T. Eldeeb, X. Xie, P. A. Bernstein, A. Cidon, and J. Yang. Chardonnay: Fast and general datacenter transactions for On-Disk databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 343–360, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/eldeeb.

[18] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 394–409, New York, NY, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815407.

[19] G. Feng, H. Cao, X. Zhu, B. Yu, Y. Wang, Z. Ma, S. Chen, and W. Chen. {TriCache}: A {User-Transparent} Block Cache Enabling {High-Performance} {Out-of-Core} Processing with {In-Memory} Programs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 395–411, 2022. ISBN 978-1-939133-28-1.

[20] M. Fleming. A thorough introduction to eBPF, 2017. https://lwn.net/Articles/740157/.

[21] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020. ISBN 978-1-939133-19-9.

[22] X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 100–115, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483556.

[23] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 440–456, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483543.

[24] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL https://doi.org/10.1145/564585.564601.

[25] B. Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[26] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 169–184, 2016. ISBN 978-1-931971-33-1.

[27] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda. Protean: {VM} Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020. ISBN 978-1-939133-19-9.

[28] J. Helt, M. Burke, A. Levy, and W. Lloyd. Regular Sequential Serializability and Regular Sequential Consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 163–179, New York, NY, USA, Oct. 2021. Associ-

ation for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483566.

[29] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 239–256, 2021. ISBN 978-1-939133-22-9.

[30] Z. Ji, K. Chen, L. Wang, M. Zhang, and Y. Wu. Falcon: Fast oltp engine for persistent cache and non-volatile memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 531–544, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613141. URL https://doi.org/10.1145/3600006.3613141.

[31] Z.-M. Jiang, S. Liu, M. Rigger, and Z. Su. Detecting transactional bugs in database engines via Graph-Based oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 397–417, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/jiang.

[32] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, Oct. 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132764.

[33] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram. WineFS: A hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 804–818, Virtual Event Germany, Oct. 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483567.

[34] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 605–620, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483548.

[35] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 424–439, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483589.

[36] K. Kostopoulou, P. Tholoniat, A. Cidon, R. Geambasu, and M. Lécuyer. Turbo: Effective caching in differentially-private databases. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 579–594, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613174. URL https://doi.org/10.1145/3600006.3613174.

[37] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 390–405, New York, NY, USA, Oct. 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132784.

[38] B. Lampson. Hints and Principles for Computer System Design, May 2021.

[39] A. Lefort, Y. Pipereau, K. Amponsem, P. Sutra, and G. Thomas. J-NVM: Off-heap Persistent Objects in Java. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 408–423, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483579.

[40] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, Oct. 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132756.

[41] H. Li, M. L. Putra, R. Shi, X. Lin, G. R. Ganger, and H. S. Gunawi. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 263–279, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483573.

[42] M. Li, X. Zhao, L. Chen, C. Tan, H. Li, S. Wang, Z. Mi, Y. Xia, F. Li, and H. Chen. Encrypted databases made secure yet maintainable. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 117–133, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/li-mingyu.

[43] X. Liao, Y. Lu, Z. Yang, and J. Shu. Crash Consistent Non-Volatile Memory Express. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 132–146, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483592.

[44] J. T. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 201–217, New York, NY, USA, Oct. 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132754.

[45] Linux. Shadow Variables, 2018. https://www.kernel.org/doc/Documentation/livepatch/shadow-vars.txt.

[46] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 819–835, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483581.

[47] X. Luo, J. Shen, P. Zuo, X. Wang, M. R. Lyu, and Y. Zhou. Chime: A cache-efficient and high-performance hybrid index on disaggregated memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 110–126, 2024.

[48] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 399–413, New York, NY, USA, Oct. 2019. Association for Computing Machinery. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359657.

[49] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 243–262, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483568.

[50] S. A. Mehdi, D. Hwang, S. Peter, and L. Alvisi. ScaleDB: A scalable, asynchronous In-Memory database. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 361–376, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/mehdi.

[51] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, Nov. 2013. Association for Computing Machinery. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522716.

[52] J. Park and Y. I. Eom. Fragpicker: A new defragmentation tool for mod-

ern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 280–294. Association for Computing Machinery, 2021. doi: 10.1145/3477132.3483593. URL https://doi.org/10.1145/3477132.3483593.

[53] S. Park, D. Zhou, Y. Qian, I. Calciu, T. Kim, and S. Kashyap. {Application-Informed} Kernel Synchronization Primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 667–682, 2022. ISBN 978-1-939133-28-1.

[54] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, 2014. ISBN 978-1-931971-16-4.

[55] S. Pirelli and G. Candea. A Simpler and Faster {NIC} Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 225–241, 2020. ISBN 978-1-939133-19-9.

[56] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 392–407, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483550.

[57] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A {User-Programmable} {SSD}. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, 2014. ISBN 978-1-931971-16-4.

[58] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 441–456, New York, NY, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815405.

[59] E. Tsalapatis, R. Hancock, T. Barnes, and A. J. Mashtizadeh. The Aurora Single Level Store Operating System. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 788–803, New York, NY, USA, Oct. 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483563.

[60] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, Oct. 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132750.

[61] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. {MemLiner}: Lining up Tracing and Application for a {Far-Memory-Friendly} Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 35–53, 2022. ISBN 978-1-939133-28-1.

[62] Q. Wang, Y. Lu, J. Li, and J. Shu. Nap: A {Black-Box} Approach to {NUMA-Aware} Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111, 2021. ISBN 978-1-939133-22-9.

[63] L. Xiang, Z. Lin, W. Deng, H. Lu, J. Rao, Y. Yuan, and R. Wang. Nomad:{Non-Exclusive} memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, 2024.

[64] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, P. Cheng, and M. Yang. Spfresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 545–561, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613166. URL https://doi.org/10.1145/3600006.3613166.

[65] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang. KnightKing: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 524–537, New York, NY, USA, Oct. 2019. Association for Computing Machinery. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359634.

[66] C. Zeng, L. Luo, Q. Ning, Y. Han, Y. Jiang, D. Tang, Z. Wang, K. Chen, and C. Guo. {FAERY}: An {FPGA-accelerated} Embedding-based Retrieval System. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 841–856, 2022. ISBN 978-1-939133-28-1.

[67] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang, M. Yang, and L. Zhou. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 377–395, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi.

[68] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, and A. Cidon. {XRP}: {In-Kernel} Storage Functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022. ISBN 978-1-939133-28-1.

[69] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. {RackSched}: A {Microsecond-Scale} Scheduler for {Rack-Scale} Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240, 2020. ISBN 978-1-939133-19-9.

Söze: One Network Telemetry Is All You Need for Per-flow Weighted Bandwidth Allocation at Scale

Weitao Wang T. S. Eugene Ng

Rice University

Abstract

Weighted bandwidth allocation is a powerful abstraction that has a wide range of use cases in modern data center networks. However, realizing highly agile and precise weighted bandwidth allocation for large-scale cloud environments is fundamentally challenging. In this paper, we propose Söze, a lightweight decentralized weighted bandwidth allocation system that leverages simple network telemetry features of commodity Ethernet switches. Given the flow weights, Söze can effectively use the telemetry information to compute and enforce the weighted bandwidth allocations without per-flow, topology, or routing knowledge. We demonstrate the effectiveness of Söze through simulations and testbed experiments, improving TPC-H jobs completion time by up to 0.59× and 0.79× on average.

1 Introduction

As a fundamental building block of the modern data center, the transport layer provides accurate and reliable data delivery between machines. Transport solutions in use by data center networks today [12, 33, 35, 62] aim to achieve a fixed goal, where the bandwidth resource sharing is generally fair and the actual allocation only depends on the traffic pattern. Unfortunately, since modern cloud applications' performance goals [3, 4, 8, 25, 43] for data transmissions are diverse and may change over time [49], such a rigid resource allocation strategy clearly cannot suit every application's need perfectly.

To better serve the various and evolving applications, weighted allocation can be a powerful abstraction [37, 44]. For example, when a flow with weight 3 shares a bottleneck link with another flow with weight 1, the bandwidth is allocated 75% and 25%. Weighted allocation has many potential use cases. For example, important jobs or latency-sensitive jobs can be prioritized for shorter waiting time [17, 26]; within a job, resources can be prioritized for critical paths to reduce the job completion time [48]; service-level objectives can be achieved by carefully assigning weights [42].

Although weighted bandwidth allocation can be extremely helpful and provide unique benefits for applications [37, 44], the reason that blocks its wide deployment is the scale of the cloud data center [10, 15]. Realizing the flow rates that conform to a weighted bandwidth allocation policy at scale is challenging. Existing systems try to implement and enforce the policies through either packet scheduling [21, 52] on switches or using a logically centralized bandwidth allocator together with rate shapers [32, 51]. On one hand, packet scheduling-based implementation is limited by the small number of per-flow queues and the coarse granularity of weight parameters supported by switch hardware [41, 61]. On the other hand, a logically centralized bandwidth allocator-based implementation has high communication latency to gather and dispatch information and high computation cost to calculate the optimal allocation [27, 32]. Thus, realizing weighted bandwidth allocation for cloud networks with fine granularity, high agility, and high scalability remains an open challenge.

In this paper, we provide all the above properties for weighted bandwidth allocation with our approach called Söze. Our insight is that we need only one network telemetry along a flow's path, regardless of the number of hops, to solve the weighted bandwidth allocation problem at the bottleneck. Furthermore, instead of using this telemetry to gauge congestion levels like in traditional use cases of network telemetry, this telemetry is cleverly repurposed to signal the correct weighted fair share at the bottleneck.

With Söze, every flow can be associated with a weight regardless of the number of flows or the size and topology of the network (scalability, generality), each flow's weight can be fine grain adjusted instantaneously at the host (granularity, flexibility), and the weighted bandwidth allocation is realized in a matter of several RTTs (agility). These capabilities provide a building block for supporting various application scenarios such as critical path prioritization, coflow straggler mitigation, intelligent bandwidth sharing across jobs, altruistic bandwidth sharing, shortest flow prioritization, etc.

Our contributions can be summarized as follows:

- Söze shows that one network telemetry is enough to reflect the network sharing status and serve as a channel

to coordinate multiple flow senders, thus offering an effective control knob for weighted bandwidth allocation.
- Söze provides a novel decentralized framework to enforce the weighted max-min fair allocation for different flows across the network with high precision and agility, without per-flow, topology, or routing information.
- We show that Söze can be seamlessly incorporated into existing transport layer mechanisms such as TCP and eRPC. In the evaluation, we show that Söze reduces the job completion time for the TPC-H benchmark up to 0.59× and 0.79× on average.

This paper is organized as follows. §2 motivates weighted bandwidth allocation and discusses challenges in designing an ideal weighted bandwidth allocation system in a large-scale cloud. §3 introduces our proposal Söze with thorough design details and proves that leveraging the network telemetry feature can achieve highly accurate weighted bandwidth allocation. §4 and §5 provide implementation details and extensive experiments to demonstrate the effectiveness of Söze. §6 discusses related work and we conclude in §7.

2 Motivation

In this section, we firstly introduce the benefits of weighted bandwidth allocation for the network (§2.1); Next, we list the essential requirements for an efficient and powerful weighted allocation solution (§2.2); Lastly, we motivate that the in-network telemetry (INT) can be a low-cost and efficient solution to achieve weighted bandwidth allocation (§2.3).

2.1 Weighted Bandwidth Allocation

Instead of a rigid bandwidth allocation strategy, weighted bandwidth allocation can give modern applications extra flexibility to adapt the underlying bandwidth allocation intentionally for better performance, which benefits many applications, like ML training, Spark/Hadoop, databases, and RPCs. With the increasing demand for giant applications [11, 55] and low-cost function services [24, 57], modern cloud applications tend to rely on the data center network for efficient data exchange under distributed execution. Take distributed training with data parallelism as an example, the tensors at different layers can be assigned different priorities for accessing the bandwidth resource, in order to overlap the communication and computation process and accelerate the overall training process [28, 29, 47]. For map-reduce applications such as Spark and Hadoop, prioritizing the straggler flow over others during the shuffle operation is a common technique to shrink the job competition time [18, 19].

2.2 Challenges

As the key infrastructure for interconnecting hardware in the cloud, one would expect the network to provide efficient and fine-grained weighted bandwidth allocation services to meet the various requirements from the applications. However, networks in the production environment only provide coarse-grained differentiation, such as traffic aggregate classes with bandwidth reservations or strict priorities. The challenges in providing efficient and fine-grained weighted bandwidth allocation services for the network come from multiple aspects:

Bottleneck resource recognition. Unlike many other computational and storage resource, the data center network has multiple layers and each layer contains multiple alternative connections, where the routing of each flow is determined with random seeds and is difficult to predict. For the flow that utilizes such networks, its path will travel multiple hops where every hop has a bandwidth capacity. On different hops, the flow will contend and share the bandwidth resources with an unpredictable group of other flows, but only one of the hops will become the bottleneck and limit its sending rate. Recognizing the bottleneck resource is therefore crucial for determining the weighted allocation of bandwidth resource but the bottleneck resource is highly unpredictable.

High scalability with large network size and high concurrency. Besides recognizing bottleneck resources, the intended system size adds another dimension of challenge: 1) Massive information: the amount of information we can collect from the network is massive, including the network topology, link bandwidth, and each flow's sender/receiver/routing. How to filter and pick the most useful information and reduce the input to the algorithm can be challenging. 2) Global optimality: since the weighted bandwidth allocation needs to be enforced at the bottleneck resource to be effective, how to reach this global optimal state with low cost and consistently fast solving time is also challenging.

Fine-grained weighted allocation. To support fine-grained policy changes, the weighted bandwidth allocation system should accommodate flow weight with fine granularity and enforce the desired allocation with high precision. However, due to the network's scale, fine granularity and high accuracy are hard to obtain cost-effectively through high-precision computations in switch ASICs. Fine-grained weighted bandwidth enforcement must require careful algorithm design instead of relying on switch hardware capability.

High agility in updates and changes. A weighted bandwidth allocation policy needs to be realized quickly, but this is challenging because the network environment dynamics can affect the allocations, and the solution must react quickly. Firstly, the network condition may change due to updates, like new flow arrivals, existing flow finishes, link failures, and routing changes. Secondly, the application may update the task weight based on the runtime information or user input.

2.3 Use Telemetry for Coordination

In-network telemetry (INT) is a common feature in switch ASICs today [5–7] that enables better network visibility by in-

serting the fine-grained switch-local information (e.g., queueing delay, timestamp, transmitted bytes, etc.) into the packet headers. The typical usage for the INT data is network monitoring, informing the network operator about the current network status, such as the queue depth or the link utilization. INT data is quantitative so that it can describe the status of the network accurately in a timely manner. In this paper, we want to use the INT feature to help achieve weighted bandwidth allocation. To start, we introduce the INT with a realistic example and demonstrate why the INT function can be used for information exchange and coordination.

Typical INT data include queueing level, link utilization, etc. As a concrete example, let us use queueing delay for demonstration: 1) Multiple senders create an incast scenario on one link, and this link collects the queueing delay as the INT and informs all senders through packet headers. 2) The queueing delay may change if any sender changes its sending rate, but the actual change of queueing depends on the behavior of all senders together. 3) All flow senders will have the same observation about the queueing delay, such as the queueing increases if every sender increases its rate. 2) and 3) imply that if flow senders change their behavior, the changes may be reflected in the queueing delay, and all the senders will sense the change at the same time. In this way, an information exchange channel can be constructed using queueing delay. Thus, we argue for a distinctly different usage for INT: using passive INT data to coordinate different hosts.

Such an information exchange channel through INT data has many unique benefits. First, the in-network telemetry feature on switches is simple and low-cost compared to other types of information exchange, which preserves deployability and scalability. Second, the in-network telemetry data is inherently quantitative, which can be used to achieve fine-grained goals with precision. Thirdly, the in-network telemetry feature can operate on a per packet granularity at full line rate, therefore timely information exchange can be achieved at very high speed. All of the above properties of in-network telemetry are favorable for designing a low-cost scalable system for bandwidth allocation, but we still need to design an algorithm to make the best use of the INT data.

3 Design

Driven by the observations in §2, we propose Söze, a decentralized weighted bandwidth allocation system for large-scale and highly dynamic cloud environments. Given the flow weights, Söze can identify the bottleneck and enforce the weighted max-min fair bandwidth allocation with only one network telemetry. In this section, we firstly show how Söze achieves the weighted allocation on a single switch, where the bottleneck is always the switch egress (§3.1); Then, we show that Söze can be applied to arbitrary data center networks, where the bottleneck can be any hop (§3.2); Lastly, we summarize our system design (§3.3) and discussions (§3.4).

3.1 A Single Switch Scenario

In this subsection, we show step-by-step how we derive a decentralized resource allocation algorithm from a straw-man algorithm. For the goal proposed in §3.1.1, we first give a simple decentralized algorithm where a lot of information needs to be collected and exchanged in §3.1.2; then we try to reduce the amount of information to be very tiny in §3.1.3; finally, we pick only one piece of information and transform it into a different format, to make telemetry easy and practical on the switches in §3.1.4.

3.1.1 Goal: Decentralized Weighted Allocation

Denote the flows on one link are $\{f_0, f_1, ..., f_n\}$, their specified weights are $\{w_0, w_1, ..., w_n\}$, and the hop bandwidth is B. After converging to weighted bandwidth allocation, the weighted fair-share should be $\frac{B}{w_0+w_1+...+w_n}$, and flow f_k's rate, r_k, should be:

$$r_k = \frac{w_k}{w_0 + w_1 + ... + w_n} \cdot B \tag{1}$$

However, calculating this weighted allocation requires obtaining the sum of all flows' weights, where a centralized controller is usually required. No matter when a new flow joins or an existing flow wants to change the weight, the centralized controller needs to be notified. Thus, this could become the bottleneck of the whole system and prevent the weighted service from being agile, accurate, and scalable.

3.1.2 Straw-man: Massive Information Exchange

A straw-man decentralized solution can be given as follows: **each flow f_i could send its weight specification w_i to every other flow**, then every flow sender is able to calculate its rate with Equation 1 directly.

However, the problem with this straw-man solution is that the communication overhead is too high for data center networks. If there are n numbers of flows on the link, the total amount of information that needs to be exchanged is $O(n)$ for each flow sender.

3.1.3 Reduce the Information Exchange

To reduce the information exchange for achieving the weighted resource allocation, Söze splits Equation 1 into two equations. And those two equations are achieved if and only if Equation 1 is achieved (proved in Appendix B of [59]):

$$\begin{cases} r_0 + r_1 + ... + r_n = B \\ \frac{r_0}{w_0} = \frac{r_1}{w_1} = ... = \frac{r_n}{w_n} \end{cases} \tag{2}$$

For the first equation in Equation 2, Söze finds that the sum of all flows' rates is the arrival rate of the link; For the second equation in Equation 2, Söze further reformats the equation

with respect to $\max(\frac{r_i}{w_i})$ and $\min(\frac{r_i}{w_i})$. Interestingly, now we are able to rewrite Equation 2 from any single flow k's view in a decentralized manner:

$$\begin{cases} link_arrival_rate = \sum_{i=0}^{n} r_i = B \\ \frac{r_k}{w_k} = \min_{i\in[0,1,...,n]}\left(\frac{r_i}{w_i}\right) = \max_{i\in[0,1,...,n]}\left(\frac{r_i}{w_i}\right) \end{cases} \quad (3)$$

If and only if every flow individually observes that Equation 3 has been achieved, the weighted allocation is achieved for all flows. Thus, potentially, **only the information about whether $\sum_{i=0}^{n} r_i = B$ and $\frac{r_k}{w_k} = \min(\frac{r_i}{w_i}) = \max(\frac{r_i}{w_i})$ are satisfied is required to be exchanged**. In this way, Söze largely reduced the exchanged information from $O(n)$ to $O(1)$ for each flow sender.

3.1.4 Conduct Information Exchange with a Convergence Algorithm and In-network Telemetry

Although the information's content is determined, how to conduct the exchange remains a problem. In this subsection, Söze leverages the in-network telemetry and designs a convergence algorithm for every flow to converge to the equilibrium in Equation 3 with zero coordination between flows.

❶ **Use the link itself for information exchange.** Previous methods usually exchange information only among servers (flow senders), either in a master-slave architecture or an all-reduce architecture. However, the information exchange among servers is always off-path and leads to extra overhead. Thus, Söze uses an on-path device for information exchange: the link. Since flows compete for the link bandwidth resource, they must travel this link, which makes it the best channel to deliver information among servers or flow senders.

A specific new technique for links: **in-network telemetry.** Söze leverages a new and common feature of the commodity switches: In-network Telemetry (INT), which allows the switch to tag some information on the packet. Specifically, Söze only uses one of the telemetry data — **queueing delay**. The queueing delay is tagged in the data packet header on the forwarding path and reflected back to the sender with the ACK packet header on the reverse path. Only the queueing delay on the forwarding path is required and it is represented by a single 2-byte field in each packet header. This operation is simple and does not require the use of programmable switches.

❷ **Queueing delay inherently implies link arrival rate.** The queueing delay's dynamic inherently reflects the link arrival rate, more specifically, the first derivative of the queueing delay is the difference between the link arrival rate and the link bandwidth. Denote the queueing delay as D, link arrival rate as R, and link bandwidth as B:

$$\frac{dD}{dt} = \frac{R-B}{B}, \forall D > 0 \quad (4)$$

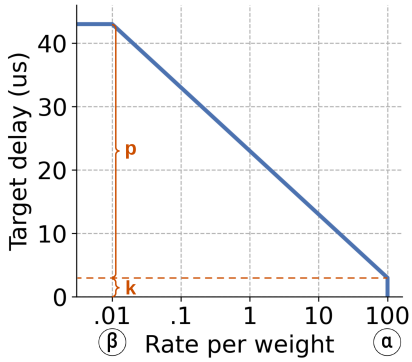


Figure 1: Target function must be monotonically decreasing. p is the queueing delay scaling and k is always > 0 for full utilization; α and β should be set to the highest and lowest rate-per-weight, depending on the scenarios.

As shown in Equation 4, as long as the queueing delay is stabilized around a non-zero value, the link arrival rate is equal to the bandwidth. Since each flow could individually observe queueing delay, the first equation in Equation 3 can be verified in a decentralized manner.

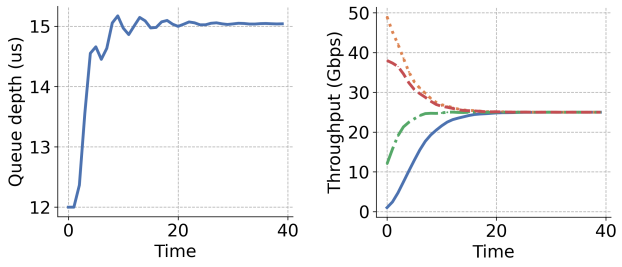
❸ **Design different queueing delays to represent different weighted fair-share.** Since the queueing delay can be stabilized around any value to indicate the link arrival rate, the actual value of the queueing delay can be used to represent another selected property. In Söze, different queueing delays represent different weighted fair-share with a unique one-to-one mapping, where the weighted fair-share wfs is defined as $wfs = \frac{r_i}{w_i}$ from Equation 2. Specifically, the larger the weighted fair share, the smaller the queueing delay. In addition, we denote a flow's rate divided by its weight as its "rate-per-weight". For networks in weighted allocation status, each flow's rate-per-weight is equal to the wfs.

$$target_delay = T(\frac{r_i}{w_i})$$
$$T(x) < T(y), \forall\, x,y \text{ where } x > y \quad (5)$$

With this relation between the queueing delay and the weighted fair share, each flow could compare its rate-per-weight with the wfs value from the queueing delay and find out if it is smaller or larger than the indicated value. In this way, a possible convergence algorithm can be created to converge to the second equation in Equation 3.

❹ **Provide a convergence algorithm that uses insights ❷ and ❸ simultaneously.** The convergence algorithm will converge to a final steady state indicated by Equation 1, where the weighted fair share is $\frac{B}{w_0+w_1+...+w_n}$, and the queueing delay on that link is a function of the weighted fair share.

The target function differentiates flows with larger $\frac{r}{w}$ from flows with smaller $\frac{r}{w}$, so that the function is monotonically-decreasing as in Figure 1. With this, each flow



(a) Convergence to the target queueing delay.

(b) Convergence to weighted fair-share.

Figure 2: Convergence for 4 flows on a single switch.

calculates a queueing delay according to their own $\frac{r}{w}$.

$$T\left(\frac{r}{w}\right) = p \cdot \frac{ln(\alpha) - ln(\frac{r}{w})}{ln(\alpha) - ln(\beta)} + k \tag{6}$$

In Equation 6, k is the minimal queueing delay when the link is saturated by one flow; p is the queueing delay scaling to differentiate between different flows. α and β indicate the most frequent range of weighted fair share: α is the highest rate-per-weight, which is usually the link bandwidth divided by the smallest weight; β is the lowest rate-per-weight, which can be determined with the traffic pattern. The update interval is referred as Δt, where $\Delta t = \frac{RTT}{CWND}$ for per-packet update.

The update function tries to change a flow's sending rate so that its target delay can match the observed queueing delay.

$$U\left(\frac{r}{w}, D\right) = \left(\frac{T^{-1}(D)}{\frac{r}{w}}\right)^{m} \tag{7}$$

In Equation 7, m is a smoother parameter smaller than 2, which is tunable to achieve either faster convergence or more stable final state.

In Algorithm 1, we give an incredibly simple decentralized algorithm for each flow sender to adjust their rate and converge to weighted allocation. After receiving the queueing delay information from the link, each flow calculates a *ratio* based on its current rate-per-weight and the received queueing delay, then updates the rate accordingly. From any initial bandwidth allocation, the final allocation will always comply with the weighted fair share as in Equation 1.

3.1.5 Proof of Convergence to Weighted Allocation

Although the above decentralized algorithm has no explicit information exchange, the queueing delay that contributed by all the flows is also observed by all the flows, which conducts an efficient information exchange towards the weighted bandwidth allocation. With the above design, we are able to have the following two lemmas:

Lemma 3.1: Convergence to Weighted Fairness

Söze converges to the weighted fairness on a link if and only if $0 < m < 2$ in the update function.

Algorithm 1: Söze's Rate Adjustment Algorithm

Input: w_k, r_k: weight and rate for flow f_k;

Function `MainFunctionForFlowK()`**:**

  $signal$ = `RecvPktWithINT()`

  if $now - last_update > rtt$ **then**

    $ratio = \mathtt{U}\left(\frac{r_k}{w_k}, signal\right)$

    $r_k = r_k \cdot ratio$

Lemma 3.2: Convergence to Target Queueing

Söze converges to the target queueing delay level, if and only if $p > \frac{\Delta t}{2} \cdot [\ln(\alpha) - \ln(\beta)]$

The idea for proving the above two lemmas is that all the flows together contribute to the queueing and maintain the queueing at the target delay level. The observed delay is the same across all flows, which provides a guidance for achieving fairness; The observed delay is maintained at a certain level, which provides the guarantee of achieving full utilization. The complete proof is in Appendix C and Appendix D of [59].

3.2 Arbitrary Network Scenario

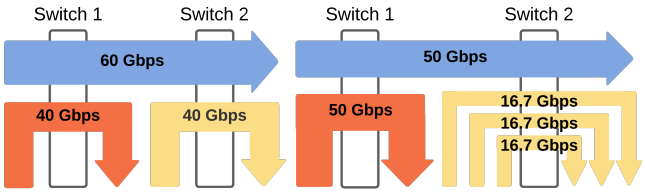
Based on the weighted allocation in a single-switch scenario (§3.1), Söze also achieves weighted bandwidth allocation in the arbitrary network scenario, such as the widely used multi-layer network architecture in production data centers. In this section, we firstly introduce the goal of the weighted resource allocation in §3.2.1; Then, to accommodate the complex scenario, we explore the properties of the flow bottleneck in §3.2.2; Lastly, we extend the collected INT signal to be the maximum queueing delay from all links on the path in §3.2.3.

3.2.1 Goal: Weighted Max-min Allocation

For the network-wide scenario, a flow may travel multiple links along the path and contend with different sets of other flows. However, for any flow traveling multiple hops, there will be at least one bottleneck link, which determines the flow rate. Thus, the weighted allocation should happen on the bottleneck link for any flow, which is also the goal of weighted max-min fair allocation. According to the previous papers, the weighted max-min fair is defined as follows:

Definition 3.1: Weighted Max-min Fair [14, 38]

For all flows $\{f1, ..., fn\}$ in the network, denote their weight to be $\{w_{f1}, ..., w_{fn}\}$. A rate allocation $\{r_{f1}, ..., r_{fn}\}$ is weighted max-min fair when for each flow f, any increase in r_f would cause a decrease in the



(a) Bottleneck for blue: Switch 1 (b) Bottleneck for blue: Switch 2

Figure 3: Bottleneck changes in weighted max-min fair: blue flow weight: 3, red flow weight: 2, yellow flow weight: 1.

transmission rate for some flow f' satisfying $\frac{r_{f'}}{w_{f'}} \leq \frac{r_f}{w_f}$.

In the weighted max-min fair, each hop divides the bandwidth to let each bottleneck flow have the same "rate-per-weight". As Figure 3 shows, the "rate-per-weight" is the same for the blue flow and the red flow, and the rate allocation is equivalent to having "weight" number of individual flows.

With the above analysis, we realize that recognizing the bottleneck becomes the new challenge when achieving the weighted max-min fair in arbitrary networks. In the next subsection, we will show how to simply recognize the bottleneck hop and obtain the INT signal from the bottleneck.

3.2.2 The Bottleneck Hop Properties

In §3.1, Söze already achieves the weighted bandwidth allocation on a single switch in a decentralized manner. Thus, in order to achieve the weighted max-min fair, the major challenge is to recognize the bottleneck hop for each flow, then achieve weighted allocation on the bottleneck hop. According to the definition, achieving fair allocation on the bottleneck hop for every flow is just achieving weighted max-min fair.

From the definition, we could derive a lemma that could help us identify each flow's bottleneck hop, namely, the hop that prevents the flow from increasing the rate further.

Lemma 3.3: Bottleneck Hop Properties

When achieving weighted max-min fair, each flow will have the largest rate-per-weight among all flows on its bottleneck hop and not on any other saturated hop.

Intuitively, a flow must have the largest rate-per-weight on its bottleneck hop, otherwise, it could take more bandwidth from the flows on this hop who has a higher rate-per-weight; a flow cannot have the largest rate-per-weight on a non-bottleneck saturated hop, since there must be some other flow on that hop could steal more bandwidth from it. The full proof of the above lemma is in Appendix E of [59].

With Lemma 3.3, we could easily know that if a flow has the largest rate-per-weight only on its bottleneck hop, then among all hops that this flow travels, the bottleneck hop must have the highest queuing delay. Because on non-bottleneck hops, there must be a flow with larger rate-per-weight, and And our target function is monotonically decreasing. Thus, the queueing delay on any other hops is lower than the bottleneck queueing delay. The proof is included in Appendix E of [59].

3.2.3 Recognize Bottleneck with *maxQD* Signal

With the above properties, we could know that a flow does not have the largest rate-per-weight on hops other than the bottleneck hop. Thus, the bottleneck hop always has the highest queueing delay for that flow. In the single-switch scenario, we use the queueing delay (*QD*) signal from one hop to achieve the weighted allocation on that hop. Similarly, for a flow that travels multiple hops in an arbitrary network, we can use the maximum per-hop queueing delay (*maxQD*) signal to achieve weighted allocation on the bottleneck hop of that flow.

The algorithm does not need to know on which hop the *maxQD* signal was collected, switches simply compare and replace the packet header to keep the *maxQD* signal is enough to achieve weighted max-min fair on the bottleneck hop. With this solution, *maxQD* becomes the signal in Algorithm 1.

With this new INT signal, we could derive our theorem on achieving weighted max-min fairness with Söze. The full proof is given in Appendix E of [59].

Theorem 3.1: Weighted Max-min Fairness

For every flow in an arbitrary network, Söze converges to a weighted max-min fair allocation, if and only if $0 < m < 2$ in the update function and $p > \frac{\Delta t}{2} \cdot [\ln(\alpha) - \ln(\beta)]$ in the target function.

3.3 System Design Summary

With all the above analysis on how to achieve weighted max-min fairness in arbitrary networks, we are able to give our system design in this section.

Collect *maxQD* in data packet and reflect in ACK packet. As shown in Figure 4, each packet contains a fixed-length packet header to store the *maxQD* INT data, which usually takes 2 bytes in the field. When the data packet travels the forwarding path, it compares and keeps the highest per-hop queueing delay on each hop. After reaching the receiver side, the receiver host will attach the *maxQD* information to the ACK packet and send it back to the sender. Thus, the *maxQD* signal is collected at the sender side for every ACK packet.

Host control with the INT signal. The rate control is conducted at the sender host. Firstly, after receiving the INT signal from the packet, the host will parse the *maxQD* data from the packet header and use it to determine the multiplicative rate update ratio. Then, the rate can be controlled with either a traditional *CWND* specification or with a rate-based specification, such as the pacing in some CC algorithms [35]. Note that, when using the window-based protocol, we are still using the "rate" rather than the congestion window directly. The

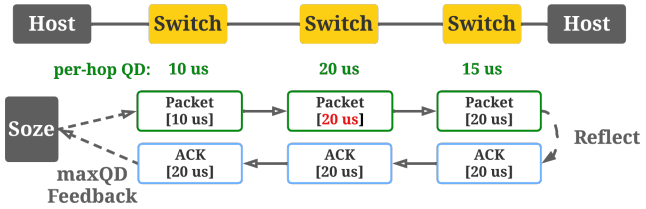


Figure 4: System design of Söze.

rate of the flow needs to be calculated with $CWND$ and RTT by $rate = \frac{CWND \cdot pktsize}{RTT}$. Lastly, Söze serves both as congestion control and as a weighted resource allocation protocol. Thus, Söze should be on the networking stack, either in the kernel or running on the NIC hardware. The applications should give the weight specification through an API.

Conduct rate update in per-packet manner. Typical CC algorithm conducts rate update for each RTT, because they are using AIMD-based algorithm for convergence. In Söze, the convergence to weighted fairness is performed with a new adaptive-update algorithm in Algorithm 1. Moreover, each individual $maxQD$ signal can reflect the network condition and calculate the rate update. Thus, achieving a per-packet rate update becomes beneficial to weight enforcement, making the convergence faster and stabilizing the rate enforcement.

3.4 Discussion

Weight distribution policy. With Söze, each application could individually determine the weight of its flows and obtain the corresponding bandwidth from the network. However, if each application increases its weight to maximum blindly, the weighted fairness will not benefit any application. For practical usage, there should be guidelines for weight determination and restrictions on the application's behavior.

The network operator could define any usage policy based on their specific workloads and service-level objectives, which, we emphasize, is not the focus of this paper. We use one simple idea for a discussion as follows: 1) Each application has a total weight that it can distribute across different hosts and flows, but the application can pay the service provider to get more total weight. 2) An application can decrease the weight of any flow when possible, namely, not fully utilize all the weights. 3) When an application wants to increase the weight of flows, some other flows in the same application need to decrease their weight so that the sum of the weights in an application is generally the same. This simple policy gives applications the ability to be altruistic to others, but when the application wants to benefit themselves, some penalties will be added to regularize their behavior.

Policy enforcement monitor. For applications inside the cloud, Söze expects them to follow the policy for weight distribution from the cloud provider. However, there is still the possibility that an application does not follow the policy. To monitor whether all the applications' behavior follows the policy, a logging system could be used. 1) Whenever the networking stack receives a weight specification from the application through the API, Söze will log the timestamp, flow ID (e.g., 5-tuple), and the specified weight. 2) Periodically, each application's weight update logs will be gathered to a policy checker, where the policy will be verified through the logs from multiple machines.

Queuing as INT signal. Although in this paper we use queuing delay as the feedback signal, the queuing level is controlled by the target function in Equation 6. Thus, the queue can be maintained at a relatively low level, as shown in Figure 13. Besides queuing delay, the design of Söze also applies to other INT signals, such as link utilization or ECN marking ratio. For different signals, the properties of the system also vary, which will be discussed in future studies.

4 Implementation

We implemented Söze with both the Linux kernel module to replace Linux's default Cubic transport and kernel-bypassing network transport — eRPC. We also provide simulator implementation on the NS-3 simulator for large-scale experiments.

Switch implementation. We implement the queueing delay signal on Tofino switches [9] with 9 lines of code. When every packet reaches the exit point of the switch, the switch uses the low-pass filter $LPF()$ function to collect the queueing signal in the "sampling mode". By controlling the sampling interval, we can adjust the signal to be each individual packet's queueing delay or smoothed queueing delay over multiple packets. In Söze, the default setting is to output the smoothed queueing delay over 10 μs.

```
1 Lpf<bit<32>, bit<10>>(size=1) lpf_queue;  # type="SAMPLE"
2 queue_input = (bit<32>) eg_intr_md.deq_timedelta;
3 queue_output = lpf_queue.execute(queue_input, 0);
```

Host implementation 1: kernel module. We implemented a prototype of Söze as a Linux kernel module with 241 lines of code, which can be installed on Linux without recompiling the kernel. By replacing the kernel module for congestion control, we could receive the maximum per-hop queueing signal in a TCP option field. For the functions $exp()$ and $log()$ in Söze, we implemented efficient approximation functions without using the STL library. For application integration, we added one TCP socket option and used that field to configure the weight for this specific TCP socket. To deliver the weight specification, we allow the application to modify the value of the TCP socket option field as the weight value. For Söze, the parameters are set to: $p = 20\mu s$, $k = 3\mu s$, $m = 0.25$. As a baseline, the default algorithm for Linux is TCP-Cubic. The switch port bandwidth is set to 25 Gbps.

Host implementation 2: eRPC. Alternatively, we also implemented Söze in an RPC library — eRPC [31], an open source RPC library that supports Ethernet, InfiniBand, and RoCE.

We add an additional field in the erpc header to carry the max queueing delay information, and use the same Tofino switches to attach those signals to each data packet. We replace the Timely CC algorithm in erpc with Söze and added supporting features, such as packet header changes and queueing signal processing, in 1972 lines of code. For application integration, eRPC uses userspace networking with polling, so applications can directly communicate with eRPC through the API and reconfigure the weight parameter for eRPC connections. The parameters for Söze are set to: $p = 20\mu s$, $k = 3\mu s$, $m = 0.25$. The bandwidth is 25 Gbps.

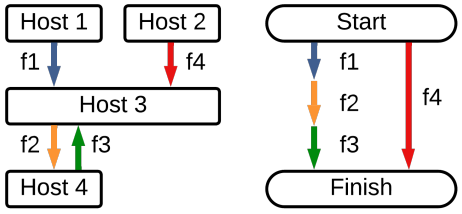
NS-3 simulator implementation. We also implement Söze in NS-3 [2]. The switches in the simulator have been customized to provide the maximum per-hop queueing delay. The packet header with "maxQD" will be updated on every hop to keep the maximum queueing delay so far. This *maxQD* information is sent back to the sender through the ACK packets. For Söze, unless otherwise noted, the default parameters in the target function are set as $p = 20\mu s$, $k = 3\mu s$, $m = 0.25$. For DCQCN, we use all the parameters suggested in [62, 63]. For HPCC, all the original settings from [35] are kept unless noted: $W_{AI} = 80$ Bytes, $maxstage = 5$, and $\theta = 95\%$.

Principles for choosing parameters. The choice of the parameters is based on the following principles: p controls the granularity of the INT signal, a higher p value leads to higher queuing delay but less rate oscillation, so we tend to choose the minimal p value that gives a relatively low rate enforcement oscillation. k controls the base queuing delay, which may cause the utilization to be less than 100% if k is too small. Thus, we choose the minimal k that provides full link utilization. m controls the convergence speed, higher m leads to faster convergence but worse rate oscillation. So, there is a wide spectrum of m that may suit different workload patterns.

5 Evaluation

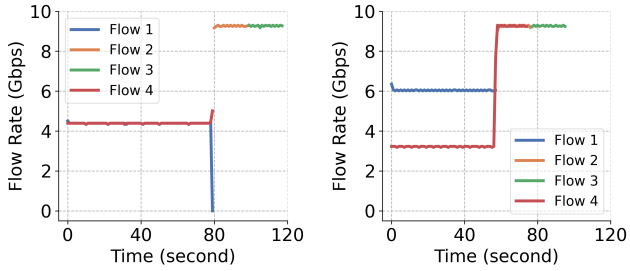
We will evaluate Söze on both the testbed and the large-scale simulator. Firstly, we demonstrate application use cases for Söze on the eRPC testbed in §5.1; Secondly, we also evaluate the efficiency of Söze as the congestion control protocol alone and compared with existing industry solutions in §5.2; Lastly, in §5.3, we conduct micro-benchmark experiments in the simulator to show that Söze can achieve fine-grained weighted max-min fair allocation rapidly on a large scale.

eRPC testbed setup. For the eRPC testbed, we connect four servers with DPDK-capable CX-4 NIC to a Tofino-1 programmable switches in a star topology. The flow is the RPC write request sent from one host to another, where the request size is 7 MB and the response size is 32 bytes. When multiple hosts send write requests to the same host, the switch egress port to that destination host will become the bottleneck. In the eRPC testbed, we compare our scheme with Timely [39], the default CC used in eRPC.



(a) Flows' src & dst. (b) Execution chart.

Figure 5: Scenario for critical path acceleration.



(a) Fair allocation. (b) Weighted: f1:f4=2:1.

Figure 6: Reduce completion time by prioritizing critical path.

NS-3 simulator setup. In the NS-3 simulator, we built a network with 1024 servers and 320 switches in a fat-tree topology. The simulator uses 100 Gbps links with 1 μs link delay, 32 MB buffer size, and 1000 bytes packet payload size. In the simulator, we compare Söze with DCQCN and HPCC. All the parameter settings are adopted from their original paper.

5.1 Application Case Study

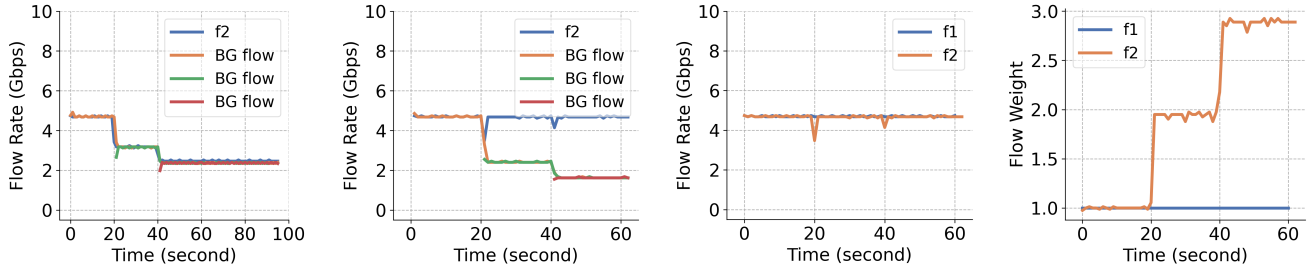
Firstly, we evaluate the application benefits from Söze in several scenarios on the eRPC testbed. The scenarios below are a small subset of the benefits for demonstration. We then evaluate Söze on the TPC-H jobs in the NS-3 simulator.

5.1.1 Prioritize Critical Path inside a Job

Consider stage-based applications such as map-reduce, distributed matrix multiplication, DNN training, etc., where there are several computation and communication phases. The end-to-end application performance depends on the performance of the critical path. Enabled with modern application-level abstractions and critical-path analysis [36, 45, 58], relative priorities between flows of different paths can be determined prior to execution, and Söze can allocate bandwidths between flows according to those priorities.

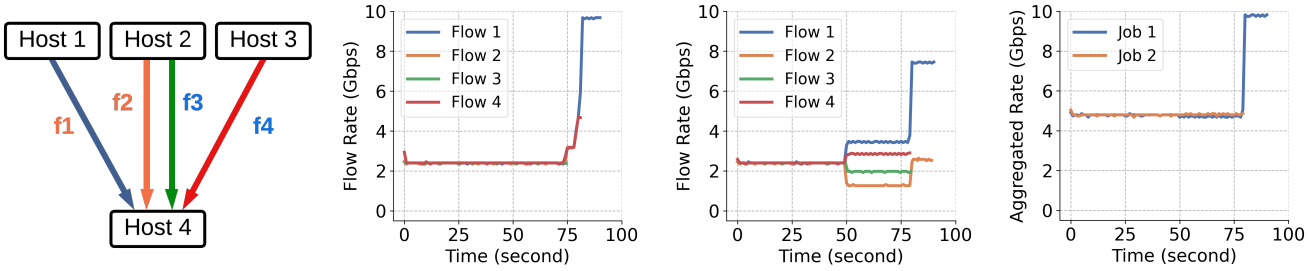
As shown in Figure 5(a), a job consists of 4 flows: flow f1 has 20 GB of data, flow f2 has 10 GB, flow f3 has 10 GB, while flow f4 has 20 GB. As shown in Figure 5(b), f2 can only start when flow f1 finishes, and f3 can only start when f2 finishes. And flow f1 and flow f4 share the same bottleneck.

When fairly sharing the resource between flow 1 and flow 4 as in Figure 6a, the start of flow 2 will be delayed, and the job completion time is 117 seconds. Once we recognize the



(a) Fair allocation for f2. (b) Straggler mitigation for f2. (c) Rates after mitigation. (d) Weights after mitigation.

Figure 7: Mitigate straggler.



(a) Experiment setup. (b) Fair allocation. (c) Weighted allocation. (d) Aggregated rates.

Figure 8: a) The flows f1 and f2 belong to job 1, and the flows f3 and f4 belong to job 2; b) The fair allocation equally allocates the bandwidth among all four flows, and rebalances when any flow finishes. Under the fair allocation, job 2 finishes at 82 seconds. c) The weighted allocation will update the weights at time 50 s, letting f3 and f4 finish at the same time and reducing the completion time for job 2 to 79 seconds; d) For the weighted allocation, each job always has the same aggregated bandwidth as before the weight changes, so that the bandwidth rebalancing only happens within a job.

critical path, we can prioritize flow 1 over flow 4 by setting their weight to be $\frac{4}{3}$ and $\frac{2}{3}$, so that flow 1 gets twice bandwidth than flow 4 and the sum of their weights still add to 2. As shown in Figure 6b, prioritized allocation reduces the job completion time to 96 seconds.

5.1.2 Identify and Mitigate Stragglers inside a Coflow

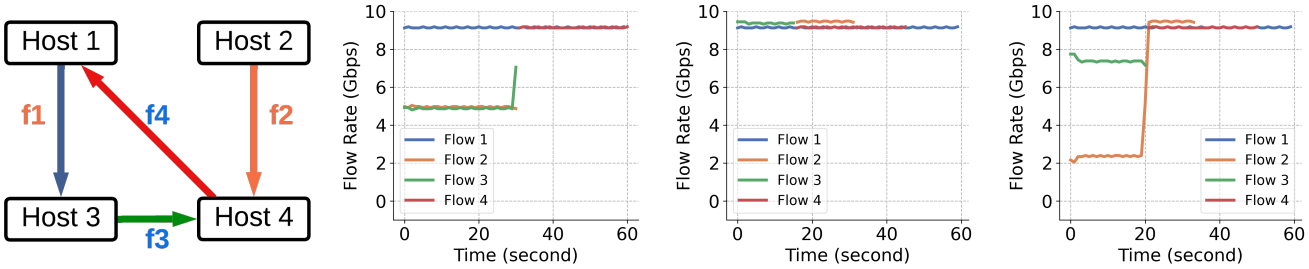
A common technique for mitigating stragglers in a coflow is to dynamically change the inter-flow priority. Such a scenario can be an interesting use case for Söze. Due to the zero-coordination and fast-convergence properties, Söze can implement the weighted bandwidth allocation with arbitrary granularity among the flows of a coflow to mitigate stragglers, and thus minimize the coflow-completion time.

This experiment tries to identify the straggler for an all-shuffle job. The figure focuses on two flows inside one coflow, where flow f2 is twice as large as flow f1. When the resources are fairly shared by f2 and background flows, flow f2 will decrease the rate as shown in Figure 7a; To mitigate the straggler and make flow f2 finish around the same time as flow f1, we monitor the progress of flow f2, if the progress cannot catch up with f1 under current sending rate, the weight of flow f2 will increase as in Figure 7d, and flow f2 will take more bandwidth from the background traffic and maintain the same rate as flow f1 as shown in Figure 7b and Figure 7c.

5.1.3 Share Resource on Common Bottleneck

Another great property of Söze is that we can reallocate the resource among flows within the same application, while the application generally keeps the same total bandwidth utilization and minimizes the effect on other applications' behavior. This property can be helpful, especially when some jobs share the same bottleneck in the network, such as the firewalls or load balancers. When applications share the same bottleneck, the resource allocation for each application can be perfectly isolated if the sum of weights remains the same.

In this experiment, we have two jobs in total. As shown in Figure 8, one job has two parallel flows f1 and f2, while the other job has two other parallel flows f3 and f4. Both jobs need data from every parallel flow for further computation, so we want them to finish at the same time to minimize waiting time. Because all the flows have different sizes, we change the weight of every flow at time 100 seconds. However, to keep the same job-level resource allocation, we choose the weight carefully so that the sum of all the flows' weights within one application remains the same. As you can see in Figure Figure 8d, no matter before or after the weight changes, the bandwidth that each job occupied in total is always around 5 Gbps, while each individual flow's bandwidth has been updated depending on the size. In summary, by restricting the sum of all flows' weights insides one job to be a constant, each job reduce the interference with other jobs' resource;



(a) Experiment setup. (b) Fair allocation. (c) Optimal allocation. (d) Altruistic allocation.

Figure 9: a) The flows f1 (28 GB) and f2 (7 GB) belong to job 1, where f1 f2 start simultaneously. The flow f3 (7 GB) and f4 (14 GB) belongs to job 2, where f4 can only started when f3 finishes; b) With the fair allocation, f2 and f3 fairly share the bandwidth and job 2 finishes at time 60 s; c) Under the optimal allocation, f2 will initially give all bandwidth to f3. Thus, job 2 finishes at time 45 s; d) With the altruistic allocation, f2 shares the bandwidth with the guarantee that it can still finish on time with its bandwidth. The job 2 finishes at around time 50 s.

Only when different jobs shares the same bottleneck, there is no interference among different jobs.

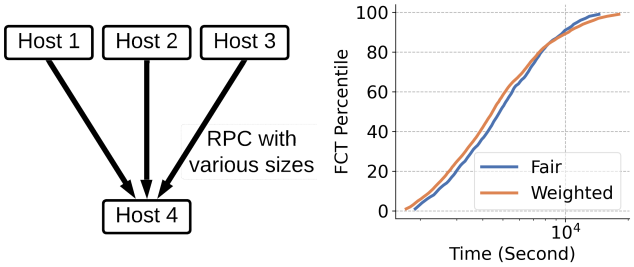
5.1.4 Altruistic Scheduling among Multiple Jobs

Although the weighted allocation could help us prioritize the critical execution path, sometimes the critical path cannot be accelerated due to resource limitations and job characteristics. In such cases, other non-critical execution paths could be altruistic by giving up some resources to other jobs, as long as the non-critical execution paths can finish at the same time as the critical path, the job completion time is not harmed. However, the bandwidth altruistically given to other jobs may benefit their completion time.

In the example of Figure 9, we have two jobs: one job has two parallel flows, f1 and f2, while the other job has two sequential flows f3 and f4. In the fair allocation case, flows f2 and f3 fairly share the inbound bandwidth at host 4, and both finish at around 30 seconds. Then, flow f4 starts and finishes at 60 seconds. However, because flow f1 is much larger than flow f2, we could let flow f2 give up some bandwidth without hurting the job completion time. In the optimal allocation scenario, flow f2 can give all the bandwidth to flow f3 and achieve the minimum job completion time. However, this is not safe because one job does not know the size of the other job's flow. Thus, in the altruistic allocation case, we let f2 get only 25% of the bandwidth, because this is the minimum bandwidth to guarantee completion at the same time with flow f1. With this altruistic behavior, flow f3 finished much earlier at 20 seconds, so that its job also finished at time 60 seconds, which is 10 seconds earlier than the fair allocation.

5.1.5 Shortest-flow Prioritization

In this experiment, we create an in-cast scenario where three hosts send RPC write requests to one host. Note that this experiment is not under the assumption that all flows come from the same job and need to follow the weight distribution policy. The size of the RPC write varies based on the workload, from



(a) Experiment setup. (b) Flow completion time.

Figure 10: Shortest flow prioritization.

around 700 MB to 7000 MB. Because the three senders may send RPCs of different sizes at the same time, prioritizing the RPCs with the smallest size could approximate shortest job first scheduling and reduce the overall average flow completion time. Thus, based on the flow size, we calculate a specific weight for flow with the equation $weight = \frac{max_flow_size}{flow_size}$.

As we can see in Figure 10, when we update the weight according to the flow size, smaller flows finish faster than the fair allocation case. As a trade-off, only a small portion of large flows suffer from longer completion time than fair allocation, while more than 80% of the flows benefit from the approximately shortest-flow first allocation. Moreover, unlike a strict shortest-flow first policy, like the preemption-based schedulers, Söze never starves the large flows.

5.1.6 TPC-H Benchmark Acceleration

Furthermore, we also test the 22 jobs from the TPC-H benchmarks [1] in our NS-3 simulator, in which each job is an execution DAG with tasks. Each task is randomly placed on a host within a fat-tree topology. For every flow in the DAG, we calculate the longest distance from this flow to the end of the DAG and use this distance to determine the flow's weight. This simple policy is designed based on the insight that the longer execution path needs more resources. In Figure 11, with Söze providing the weighted allocation, the average job completion time is reduced by 0.79×, and the maximum reduction is 0.59×. Only jobs that have only one execution path

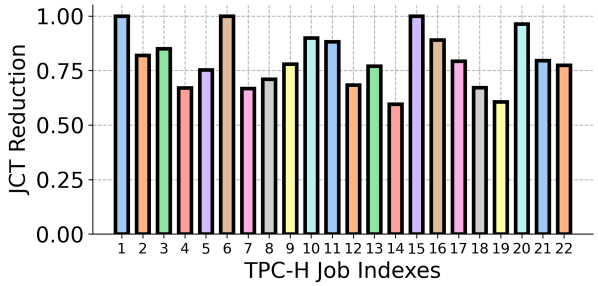
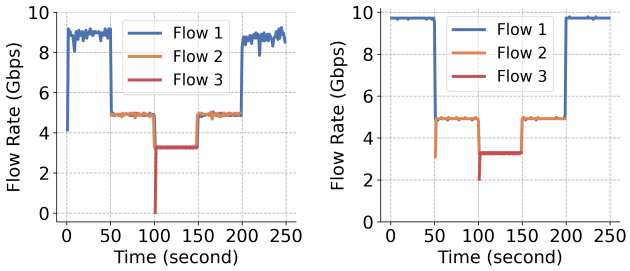


Figure 11: TPC-H jobs acceleration with weighted allocation.



(a) Timely. (b) Söze.

Figure 12: Step-in & step-out experiment on eRPC testbed. Söze achieves higher utilization and faster convergence to the optimal state than Timely.

do not receive benefits from Söze.

5.2 Efficient Congestion Control

5.2.1 Step-in & Step-out Experiment in Testbed

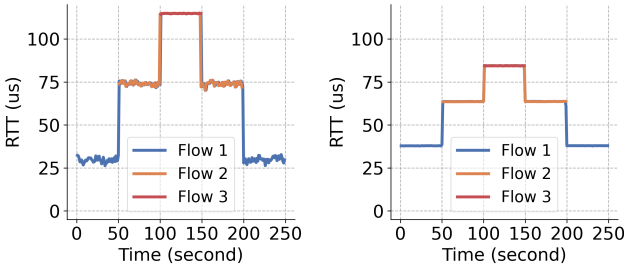
To show that Söze can serve as an efficient congestion control algorithm, we conduct a step-in and step-out experiment on the eRPC testbed with three hosts sending traffic to one host. Every 50 seconds, a new flow was added to the incast host from a different sender until time 100 seconds, then every 50 seconds, one flow will terminate and give bandwidth back.

As shown in Figure 12, Söze outperforms Timely, the default congestion control algorithm in eRPC. When there is only one flow, Söze achieves higher utilization than Timely; and when a new flow arrives or an existing flow completes, Söze converges to the new optimal allocation faster than Timely because of the proposed adaptive MIMD algorithm.

In Figure 13, we also show the round-trip time (RTT) for both Timely and Söze. The RTT of Söze increases with more number of flows competing on the link, because our target function is monotonically decreasing as the fair-share rate of the link increases. Nevertheless, Söze still achieves lower RTT than Timely, because we use a range of queueing delay to indicate different levels of congestion, and the maximum queueing delay in Equation 6 bounds the RTT of Söze.

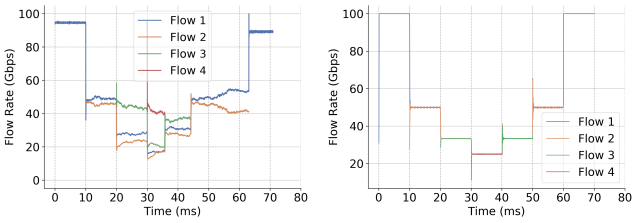
5.2.2 Step-in & Step-out Experiment in NS-3

In the NS-3 simulator, we also conducted a step-in and step-out experiment with hosts under different ToRs sending traffic



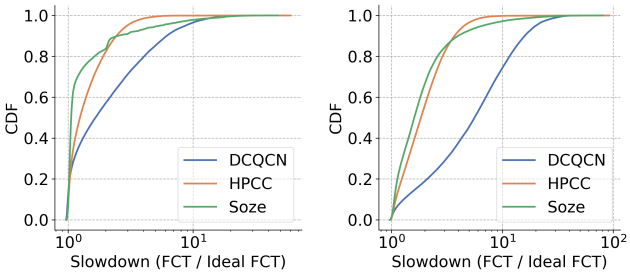
(a) Timely. (b) Söze.

Figure 13: Söze achieves lower RTT than Timely by using queueing level as the signal to indicate congestion.



(a) HPCC. (b) Söze.

Figure 14: Step-in & step-out experiment in NS-3 simulator. Compared with HPCC, Söze achieves a more stable and accurate rate allocation.



(a) 40% load. (b) 80% load.

Figure 15: FCT slowdown under different network loads.

to the same destination. We compared Söze with one recent data center congestion control algorithm using in-network telemetry — HPCC. In this set of experiment, 4 flows are added every 10 milliseconds, and each flow completes every 10 milliseconds in sequence.

Compared to HPCC, the bandwidth allocation in Söze is very accurate and stable in Figure 14. Because Söze uses the queueing delay as the signal to indicate the level of congestion, more specifically, the fair-share rate of the link. While for HPCC, the in-network signal is only used to indicate whether the link is congested or not, so that the bandwidth allocation is not accurate nor stable for HPCC due to the binary status provided by its in-network signal.

5.2.3 FCT Slowdown

In order to evaluate the performance of Söze under high network load with a large amount of flows, we run the Google RPC workload [53] with randomly selected sender and re-

ceiver. And we compare with two datacenter congestion control solutions — HPCC and DCQCN.

In Figure 15, under both 40% and 80% load, Söze achieves lower FCT slowdown than HPCC and DCQCN, especially for the short flows. Because Söze could grab bandwidth rapidly with fast convergence. As shown in Figure 16, Söze provides more benefits for flows with less than 1000 packets. While for flows with larger data size (> 100k packets), Söze also provides comparable performance with DCQCN and outperforms HPCC, leading to shorter tail FCT slowdown in Figure 15.

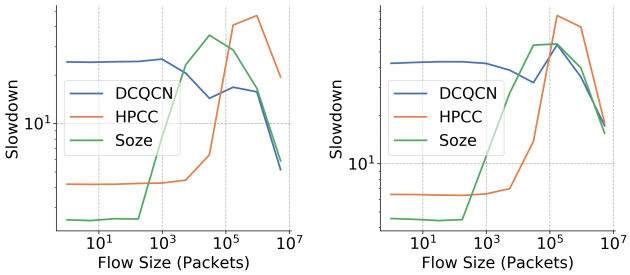
5.3 Micro-benchmark

In this subsection, we compare Söze with alternative solutions with a set of micro-benchmark experiments on the NS-3 simulator to demonstrate Söze's weighted max-min fairness, high agility, fine granularity, and scalability.

5.3.1 Weighted Max-min Fairness

In the experiment for weighted max-min fairness, we create a scenario where flow 1 only travels switch 1; flow 2, 3, and 4 travel switch 1 and switch 2; flow 4 and 5 only travel switch 2. The link bandwidth is 100 Gbps and the default weight for every flow is 1, so that flow 1 get 40 Gbps at the beginning. Every 10 milliseconds, the weight of flow 1 will be increased by 1. When the weight is 2, flow 1 still gets 40 Gbps because the bottleneck hop for flow 2, 3, and 4 is still switch 2. Only when the weight increases to be higher than 3, the bottleneck hop becomes switch 1 and flow 1 will take more bandwidth from other flows on switch 1.

As we described in §3.2, the weighted max-min fairness for every flow is determined by its bottleneck hop. In Figure 17, we show that the bottleneck hop for flow 2, 3, and 4 changes with the increase of flow 1's weight. With this micro-benchmark experiment, we demonstrate that Söze can recognize the bottleneck hop for every flow and achieve weighted max-min fairness rapidly after the weight is changed for any flow in the network. Moreover, the bottleneck hop changes at time 20 ms, and Söze handles the changing bottleneck rapidly within 10 RTTs. For comparison, we test how weighted round-robin (WRR) scheduling in switches combined with AIMD rate control would perform in this scenario. WRR enforces weighted sharing of bandwidth at the packet scheduling level but senders must still discover the achievable bandwidth at the end-to-end level. The AIMD control uses queuing delay as the explicit feedback signal: when the queuing delay is below 20 μs (this value is chosen to ensure good link utilization in the experiment), the CWND is increased by 1; when the queuing delay exceeds 20 μs, the CWND is reduced by 20% (this value is chosen to limit oscillation). In the results in Figure 17c, we can see that the rate for each flow is maintained correctly around the weighted max-min fair allocation by WRR scheduling. However, the link utilization is lower than with Söze because the oscillation due to AIMD is larger.



(a) 40% load. (b) 80% load.

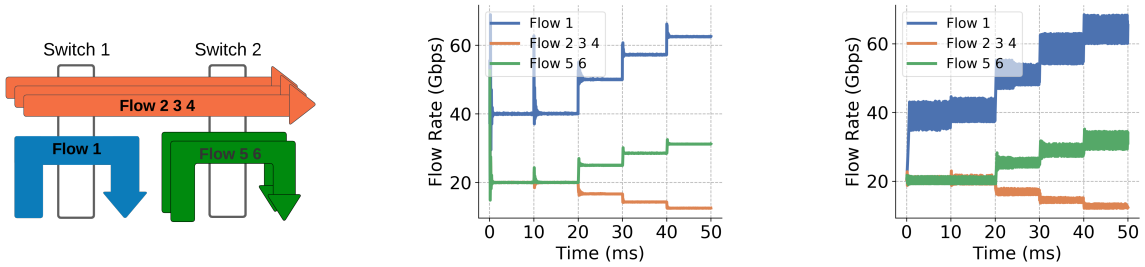
Figure 16: FCT slowdown for different flow size under different network loads.

5.3.2 Granularity Micro-benchmark

To show the granularity of the weighted allocation, we make use of a coflow scenario, where each coflow is consist of 10 flows, and the flow size is randomly chosen from a range of [1 GB, 3 GB]. In Söze, according to the size of each flow, the flow will be assigned with a certain weight to complete at the same time with other flows: in Söze's scenario, the weight is equal to the flow size. For the multi-connection solution, we will choose the closest integer number with the flow size. For instance, there will be 2 connections for a flow with 2.4 GB data size and there will be 3 connections for a flow with 2.6 GB data size. For the switch-based weighted round-robin solutions, we test on a scenario where each switch has 4 physical queues. We try to pack the 10 different flows into 4 physical queues and set a certain weight for each physical queue. The flows with similar data size will be packed into the same queue, and they will fairly share the bandwidth from that physical queue. The sum of the flow data size within a queue will be used to calculate the weight of each queue.

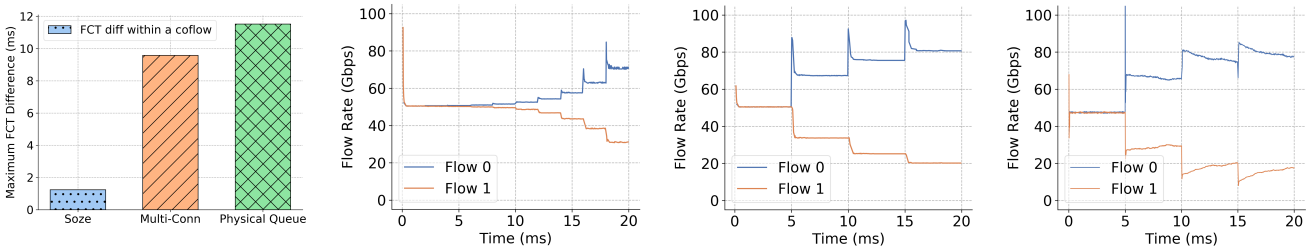
In Figure 18a, Söze allows all the flows within a coflow to finish at nearly the same time, so that it has the minimum FCT difference among all the alternative solutions. In contrast, the multi-connection solution ("Multi-Conn") only provides integer number of concurrent connections, which makes the weight enforcement rigid and cannot support non-integer weight; the switch-based weighted round-robin ("Physical Queue") relies on limited number of physical queues, and weight enforcement granularity will degrade when the number of flows exceeds the number of queues.

In addition, we also compare different schemes by changing weights or the number of connections. We create a simple in-cast scenario, where two flows are sent to the same destination host. In Figure 18b, we increase the weight for flow 0 every 2 milliseconds by 5‰, 10‰, 20‰, etc. The figure shows that the rate allocation can be accurately stabilized around any level, even when the weight difference is relatively small. In contrast, we also let both Söze and HPCC have multiple connections to change the rate allocation. In Figure 18c, Söze adds one connection for flow 0 every 5 milliseconds. The rate allocation changes accurately accordingly



(a) Experiment scenario. (b) Söze with increasing weight. (c) WRR with AIMD.

Figure 17: a) Flow 1 and flow 2, 3, 4 compete on switch 1; flow 5 and 6 compete with flows on switch 2. During time 0 ms to time 50 ms, we increase the weight of flow 1 from 1 to 5 every 10 ms. b) Söze achieves accurate weighted sharing. When the weight of flow 1 is smaller than 2, flow 1 remains at 40 Gbps since switch 2 is the most congested hop; but when the weight is higher than 2, flow 1's rate increases because the congested hop has become switch 1. c) For comparison, we repeat the experiment using weighted round-robin (WRR) scheduling in switches to achieve flow 1's increasing weight; AIMD rate control is used to adapt to the weight changes.



(a) Granularity comparison. (b) Söze with weight updates. (c) Söze multi-connection. (d) HPCC multi-connection.

Figure 18: Granularity and agility comparison.

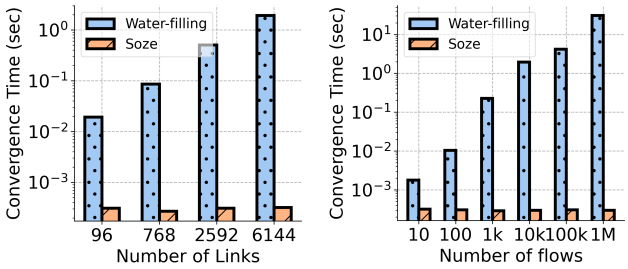
to the number of connections, but the allocation granularity is limited to weights with integer numbers. For HPCC with multi-connection in Figure 18d, the flow rates are also maintained around the weighted allocation, but the rate enforcement is less accurate and also suffers from coarse-granularity due to integer weights.

5.3.3 Agility Micro-benchmark

With the same experiment in 18b that changes the weights for Söze, we also demonstrate the agility achieved by Söze. As shown in Figure 18b, on average, it only takes less than 10 RTTs for Söze to converge to the new rate under the new weight. We can observe that there is a rate surge right after each weight change and before convergence. This is because we are increasing the weight of flow 0, thus the new converged queueing delay is higher than before, so the rate needs to exceed the line rate briefly to stack the switch queue.

5.3.4 Scalability Micro-benchmark

Scalability is the key to supporting large-scale systems with consistent performance despite increasing scale. Söze is a fully distributed system and can inherently scale to datacenters of any size; in contrast, the water-filling algorithm is a centralized solver, which needs to gather all the information to one master node and use its resources for computation. In the following experiments, we test the convergence time of



(a) Increasing topology size. (b) Increasing flow count.

Figure 19: The solving time for the water-filling algorithm increases drastically with either increasing topology size or increasing number of flows; while Söze provides consistent convergence speed toward the global weighted allocation state despite of the increasing topology size or flow count.

Söze and the solving time of the water filling to demonstrate the capability to scale for those two solutions.

Firstly, we maintain the same number of flows and change the fat-tree network topology sizes by parameter K. We chose the number of flows to be 10k so there are roughly 10 flows per server when $K = 16$. In Figure 19a, when the topology size increases from $K = 4$ (16 hosts) to $K = 16$ (1024 hosts), the solving time of the water filling algorithm increases drastically. In contrast, for Söze, once applications have determined the weights, the convergence time towards the global weighted bandwidth allocation is always around the same magnitude.

The average convergence time for Söze is 0.3 milliseconds, which is around 10 RTTs.

Moreover, we also tested the convergence time for Söze and the water filling algorithm with increasing number of flows under the same topology. As shown in Figure 19b, the network topology is always a fat-tree with 1024 servers ($K = 16$), and the number of flows in the network increases from 10 flows to 1 million flows. As more and more flows are added to the network, the solving time for the water filling algorithms increases from 1.81 milliseconds with 10 flows to 31.1 seconds with 1 million flows. While Söze still provides consistent convergence towards the global weighted allocation within 10 RTTs. Thus, the water-filling algorithm has inherent drawbacks to support large-scale data center; while Söze benefits from the decentralized design and can be scale up with nearly no performance degradation.

6 Related Work

One line of work focuses on using weighted fair queueing (WFQ) on the switches to achieve weighted allocation. NumFabric [42] assumes switches support WFQ in hardware, then provides algorithms for setting the weight parameters in switches to optimize for a utility objective. NumFabric is complementary to Söze, because Söze replaces switch WFQ; weight parameters provided by NumFabric can be used to implement the desired utility objective on Söze even more easily, because there is no talking to switches' control planes.

Besides using weighted fair-queueing algorithms on switches [16, 21, 23, 46, 52] to allocate bandwidth, a recent trend is to use programmable switches to approximate weighted fair-queueing [22, 50, 54, 60, 61]. Such switch-based solutions are hard to scale because of the resource limitations on the switches. To enforce the per-flow weighted allocation accurately among many flows, the switch data plane needs to keep a large amount of information, which may exceed the memory capacity. Moreover, the delay and the complexity of the control plane for such a scheme are both high, which leads to a lack of agility. To add a new flow or to change the weight, the flow sender needs to inform all the switches along the flow's path individually. Besides, those works heavily rely on programmable switches, which are not widely used in production, and those switches are usually less cost-effective than commodity switches with fixed telemetry functions. PERC [30] proposes an alternative approach that departs from the existing DCN service model; it requires switches to run a distributed algorithm to continuously compute the max-min fair shares for all flows and communicate them to end hosts; the algorithm requires control packets that must be processed by switches, requiring multiplication and division operations. In contrast, Söze only requires INT from the switches, and end hosts can do the rest.

As for solutions that do not rely on switches, a bandwidth allocator [27, 32, 34, 51] is also capable of achieving weighted max-min fairness for all flows. By aggregating the sources, destinations, and demands of all flows, the allocator can calculate a rate allocation plan for each flow. However, this solution requires aggregating information and calculating the allocation plan at the controller, which may lead to a long solving time when the scale of the network increases. Moreover, such solutions are not agile. Each time a flow needs to change weight or a new flow arrives, the whole rate allocation algorithm may need to be executed, leading to high overhead and long control plane delay.

There are also proposals that aim to manipulate the congestion window to allow a flow to take up more bandwidth when needed. For example, D2TCP [56] proposes to adjust the congestion window size more or less aggressively based on how close the time is to the deadline. MulTCP [20] heuristically lets a flow act like N flows in adjusting its congestion window more aggressively. Both solutions build on top of the TCP algorithm, but suffer from the inherited drawbacks, like slow convergence and throughput jitter. Weighted allocation is not the goal of D2TCP, while in MulTCP the effect of increasing the N parameter is not linear and may change for different situations. In contrast, Söze achieves fast convergence, low throughput jitter, and accurate weighted fair allocation.

Besides weighted allocation, priority-based allocation solutions [13, 40] are also used to provide service differentiation between flows, but such solutions have some intrinsic disadvantages over weighted bandwidth allocation, such as head-of-line blocking if preemption is not allowed, and starvation risk. Moreover, priority-based differentiation schemes are a larger departure from the existing DCN service model that's based on fair allocation. Söze represents a potentially more compatible evolutionary direction with flexible and controllable allocation for general-purpose DCN.

7 Conclusion

In this paper, we propose Söze, a simple and efficient weighted bandwidth allocation system for data center networks. Söze designs the maxQD INT signal to not only indicate network status but also provide a control knob to coordinate multiple flow senders. Furthermore, Söze co-designs the decentralized rate update algorithm with the maxQD signal, which allows each flow to independently move toward the weighted max-min fair allocation in arbitrary networks. We prototype Söze across various platforms and demonstrate that Söze can achieve weighted bandwidth allocation with fine granularity, high agility, and high scalability.

Acknowledgment

We thank the reviewers and the shepherd Ratul Mahajan for their insightful feedback. This work is partially supported by the NSF under CNS-2214272.

References

[1] Tpc benchmark h, 2001. http://www.tpc.org/tpch/.

[2] Ns-3, 2023. https://www.nsnam.org/.

[3] Accelerate ai development with google cloud tpus, 2024. https://cloud.google.com/tpu/.

[4] Google tpu: Architecture and performance best practices, 2024. https://www.run.ai/guides/cloud-deep-learning/google-tpu.

[5] In-band Network Telemetry in Barefoot Tofino, 2024. https://www.opencompute.org/files/INT-In-Band-Network-Telemetry-A-\PowerfulAnalytics-Framework-for-\your-Data-Center-OCP-Final3.pdf.

[6] In-band Network Telemetry in Broadcom Tomahawk 3. https://www.broadcom.com/company/news/product-releases/2372840, 2024.

[7] In-band Network Telemetry in Broadcom Trident3. https://www.broadcom.com/blog/new-trident-3-switch-delivers-\smarterprogrammability-for-enterprise-\and-service-provider-datacenters, 2024.

[8] Nvidia rtx server: Powering the future of cloud gaming and ar/vr, 2024. https://www.nvidia.com/en-us/data-center/rtx-server-gaming/.

[9] Tofino switches by intel, 2024. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors.html.

[10] Dennis Abts and John Kim. *High performance datacenter networks: Architectures, algorithms, and opportunities*. Springer Nature, 2022.

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

[12] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[13] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[14] Miriam Allalouf and Yuval Shavitt. Centralized and distributed algorithms for routing and weighted max-min fair bandwidth allocation. *IEEE/ACM Transactions on networking*, 16(5):1015–1024, 2008.

[15] Luis Andre Barroso and Jimmy Clidaras. *The datacenter as a computer: An introduction to the design of warehouse-scale machines*. Springer Nature, 2022.

[16] Jon CR Bennett and Hui Zhang. Wf/sup 2/q: worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, volume 1, pages 120–128. IEEE, 1996.

[17] Daniel Casini, Paolo Pazzaglia, Alessandro Biondi, Marco Di Natale, and Giorgio Buttazzo. Predictable memory-cpu co-scheduling with support for latency-sensitive tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[18] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[19] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM*, 2014.

[20] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *ACM SIGCOMM Computer Communication Review*, 28(3):53–69, 1998.

[21] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[22] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H Jonathan Chao. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 551–565, 2022.

[23] S Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of INFOCOM'94 Conference on Computer Communications*, pages 636–646. IEEE, 1994.

[24] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

[25] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand,

and Jon Crowcroft. Queues don't matter when you can jump them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, 2015.

[26] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[27] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, pages 1–12, 2010.

[28] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems*, 1:418–430, 2019.

[29] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *Proceedings of Machine Learning and Systems*, 1:132–145, 2019.

[30] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. High speed networks need proactive congestion control. In *Proceedings of the 14th acm workshop on hot topics in networks*, pages 1–7, 2015.

[31] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[32] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 1–14, 2015.

[33] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020.

[34] Terry Lam, Sivasankar Radhakrishnan, Amin Vahdat, and George Varghese. Netshare: Virtualizing data center networks across services. 2010.

[35] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[36] Jie Liang, Kenli Li, Chubo Liu, and Keqin Li. Joint offloading and scheduling decisions for dag applications in mobile edge computing. *Neurocomputing*, 424:160–171, 2021.

[37] Tinghuai Ma, Ya Chu, Licheng Zhao, and Otgonbayar Ankhbayar. Resource allocation and scheduling in cloud computing: Policy and algorithm. *IETE Technical review*, 31(1):4–16, 2014.

[38] Peter Marbach. Priority service and max-min fairness. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 266–275. IEEE, 2002.

[39] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[40] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[41] Sung-Whan Moon, Jennifer Rexford, and Kang G Shin. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on computers*, 49(11):1215–1227, 2000.

[42] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 188–201, 2016.

[43] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 13), pages 385–398, 2013.

[44] Houria Oudghiri and Bozena Kaminska. Global weighted scheduling and allocation algorithms. In *Proceedings The European Conference on Design Automation*, pages 491–492. IEEE Computer Society, 1992.

[45] Rui Pan, Yiming Lei, Jialong Li, Zhiqiang Xie, Binhang Yuan, and Yiting Xia. Efficient flow scheduling in distributed deep learning training with echelon formation. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 93–100, 2022.

[46] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 1(3):344–357, 1993.

[47] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *ACM SOSP*, 2019.

[48] Reza Ramezani. Dynamic scheduling of task graphs in multi-fpga systems using critical path. *The Journal of Supercomputing*, 77(1):597–618, 2021.

[49] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[50] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, 2018.

[51] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[52] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.

[53] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[54] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

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

[56] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[57] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182. IEEE, 2016.

[58] Weitao Wang, Sushovan Das, Xinyu Crystal Wu, Zhuang Wang, Ang Chen, and TS Eugene Ng. Mxdag: A hybrid abstraction for emerging applications. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 221–228, 2021.

[59] Weitao Wang and T. S. Eugene Ng. Söze: One network telemetry is all you need for per-flow weighted bandwidth allocation at scale, 2025. https://arxiv.org/abs/2506.00834.

[60] Liangcheng Yu, John Sonchack, and Vincent Liu. Cebinae: scalable in-network fairness augmentation. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 219–232, 2022.

[61] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 179–193, 2021.

[62] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

[63] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 313–327, 2016.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Decouple and Decompose: Scaling Resource Allocation with DEDE

Zhiying Xu[†] Minlan Yu[†] Francis Y. Yan[𝕀]
[†]*Harvard University* [𝕀]*University of Illinois Urbana-Champaign*

Abstract

Efficient resource allocation is essential in cloud systems to facilitate resource sharing among tenants. However, the growing scale of these optimization problems have outpaced commercial solvers commonly employed in production. To accelerate resource allocation, prior approaches either customize solutions for narrow domains or impose workload-specific assumptions. In this work, we revisit real-world resource allocation problems and uncover a common underlying structure: the vast majority of these problems are inherently *separable*, i.e., they optimize the aggregate utility of individual resource and demand allocations, under separate constraints for each resource and each demand. Building on this observation, we develop DEDE, a scalable and theoretically rooted optimization framework for large-scale resource allocation. At the core of DEDE is a *decouple-and-decompose* approach: it decouples entangled resource and demand constraints and thereby decomposes the overall optimization into alternating per-resource and per-demand subproblems that can be solved efficiently and in parallel. We have implemented and released DEDE as a Python package with a familiar modeling interface. Our experiments on three representative resource allocation tasks—cluster scheduling, traffic engineering, and load balancing—demonstrate that DEDE delivers significant speedups while generating higher-quality allocations.

1 Introduction

Resource allocation remains a critical and challenging problem today, especially as cloud providers operate multi-tenant systems on an unprecedented scale—these systems must ensure efficient and fair allocation of computing, storage, and networking resources across a large number of clients, in order to maintain the quality of cloud services.

Common scenarios of resource allocation include cluster scheduling [28,45,47,63,66], traffic engineering [1,24,32,51,69], and load balancing [14,38,49,53]. To meet diverse and dynamic user demands (jobs, traffic flows, and queries), these systems extensively employ commercial optimization solvers like Gurobi [22] to solve linear programs (LP) or mixed-integer linear programs (MILP) that determine the allocation of valuable cloud resources (compute units, network links, and data nodes) worth millions to billions of dollars.

However, as cloud environments continue to expand and diversify, the sheer scale of modern resource allocation problems has exceeded the capabilities of commercial solvers.

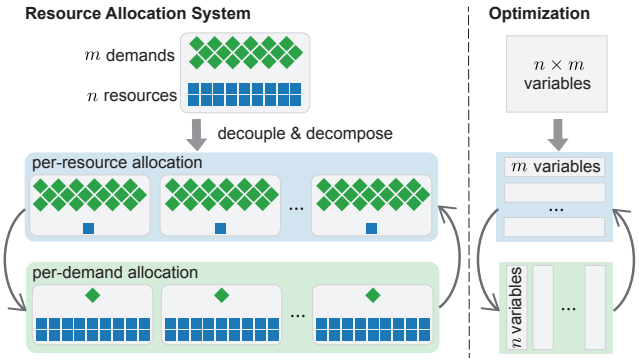


Figure 1: Overview of DEDE.

These problems may involve millions of variables and take tens of minutes or even hours to solve, whereas fast allocation is necessary to meet service-level objectives. This "scalability crisis" has spurred a flurry of studies [1,43,48,65] that trade off solution quality for computational speed, using heuristics, approximate algorithms, or machine learning. Nevertheless, these approaches are typically restricted to specific domains (e.g., wide-area network traffic engineering) or particular optimization objectives (e.g., max-min fairness), limiting their broader applicability. While POP [44] accelerates assorted allocation tasks, it hinges on a "granular" assumption that each demand requests only a small fraction of interchangeable resources. This premise proves brittle under realistic workloads, leading to degraded solution quality (§7).

In this study, we revisit real-world resource allocation problems through a different lens, avoiding domain-specific designs or workload-dependent assumptions. We introduce DEDE[1], a practical and scalable optimization framework for solving resource allocation problems. Rooted in optimization theory, DEDE allows the global allocation problem to be decomposed into alternating per-resource and per-demand subproblems (Figure 1), which can be solved more efficiently and in parallel without compromising solution quality.

This decomposition is enabled by an intrinsic, workload-agnostic problem structure that we identified after surveying dozens of real-world resource allocation problems in the literature (Table 1). We find that the vast majority of these problems are inherently *separable*: they optimize the total utility of individual resource and demand allocations, subject to separate constraints for each resource and demand to ensure feasibility. To be more concrete, we illustrate this problem structure using a simplified job scheduling example.

[1] DEDE encapsulates our core technique "**De**couple and **De**compose."

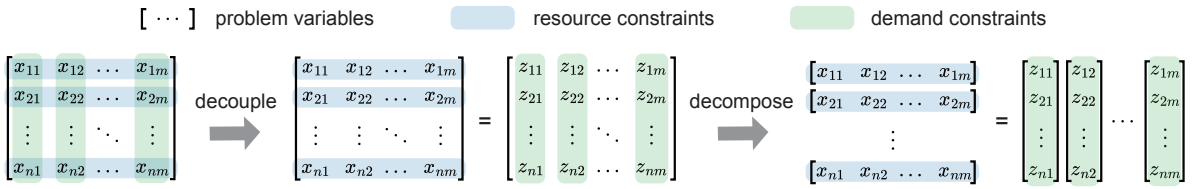


Figure 2: DEDE decouples constraints using an auxiliary variable, thereby decomposing optimization into subproblems.

▶ **Example:** Each hour, we aim to determine—as soon as possible—the optimal allocation of a set of large language model inference jobs (e.g., GPT-4, Llama 3, and DeepSeek-V3) on a heterogeneous GPU cluster composed of various hardware types (e.g., Nvidia H100, A100, and V100). Jobs may divide their execution time across multiple GPU types, and due to differences in hardware specifications (e.g., memory capacity, bandwidth, and FLOPS), each job may require a different number of GPUs for each GPU type and achieve varying levels of throughput, measured in tokens per second (TPS). For instance, a job might run for 0.4 hours on 2×A100 at 50 TPS, and for 0.6 hours on 4×V100 at 20 TPS.

Formally, let x_{ij} denote the fraction of the hour that job j (demand) runs on GPU type i (resource). Naturally, we have the demand constraints: $\forall j, \sum_i x_{ij} \leq 1$. On the other hand, for each GPU type, the total GPU hours requested by all jobs must not exceed the available GPU capacity. Thus, the resource constraints are: $\forall i, \sum_j \text{req}_j \cdot x_{ij} \leq \text{capacity}_i$. The objective is to maximize the total weighted average throughput, $\sum_j w_j \cdot (\sum_i \text{tput}_{ij} \cdot x_{ij})$, reflecting different job priorities. □

This example demonstrates the "separable" structure frequently encountered in practical resource allocation problems: resource constraints are specified for *each* resource and its associated demands, demand constraints are specified for *each* demand and its contributing resources, and the objective function sums the utility achieved by *each* demand (or resource). Despite this seemingly modular structure, however, such optimization problems remain computationally challenging for commercial solvers to solve at scale. The difficulty arises from each allocation variable x_{ij} simultaneously affecting both resource and demand constraints, precluding straightforward decomposition and parallel optimization.

To address this challenge, DEDE first *decouples* entangled resource and demand constraints through a mathematical reformulation. It introduces a copy of the allocation matrix x as an auxiliary variable z, and rewrites the demand constraints (and per-demand terms in the objective) in terms of z, while adding a new constraint $x = z$. This transformation is mathematically equivalent, preserving the optimal solution. Moreover, it also lends itself to ADMM (Alternating Direction Method of Multipliers) [8], a constrained optimization method based on Lagrange multipliers [6]. By incorporating constraints into the objective via multipliers and constructing the (augmented) Lagrangian (see §3.1), ADMM provides a theoretical framework that allows DEDE to alternatively and iteratively optimize the Lagrangian with respect to each block of variables—x or z—while holding the other fixed.

Building on the reformulation, DEDE further *decomposes* the optimization into subproblems along each resource and each demand. This step capitalizes on the identified separable structure, wherein the introduced Lagrangian can be partitioned across individual resources and demands (see §3.2). Consequently, instead of solving a monolithic optimization over all n resources and m demands, DEDE breaks the problem into n per-resource subproblems, each with only m demand variables, and m per-demand subproblems, each with only n resource variables. These smaller subproblems are amenable to independent and parallel optimization using off-the-shelf solvers. Figure 2 depicts the decouple-and-decompose workflow of DEDE.

To demonstrate DEDE's practical utility, we have implemented DEDE as a Python package, installable via `pip install dede`. Built on the popular open-source modeling language `cvxpy` [3, 15], DEDE provides familiar APIs (§6) aligned with `cvxpy`. Moreover, unlike prior efforts that simulate parallelism [1, 44], our implementation is optimized for *real* parallel execution using Ray [41], enabling efficient utilization of many CPU cores with minimal overhead.

We evaluate DEDE on three representative resource allocation tasks—cluster scheduling, traffic engineering, and load balancing, demonstrating its faster speed and higher allocation quality relative to the state of the art (§7). Compared with the best variant of POP [44] in each domain, DEDE achieves a 7.3% improvement in allocation quality and a 3.1× speedup for cluster scheduling, 5.3% and 7.6× for traffic engineering, and 12.6% and 2.2× for load balancing.

2 Real-World Resource Allocation Problems

In modern cloud systems, resources such as CPUs, memory, and network bandwidth are shared among numerous users and applications. A central resource allocator often casts the distribution of resources as an optimization problem (e.g., LP or MILP) and repeatedly computes feasible solutions using solvers to accommodate changing demands.

However, these solvers are struggling to keep up with the increasing problem sizes that frequently occur in fast-growing systems. They may take up to *several hours* to solve an allocation problem that attempts to assign thousands of resources to thousands of demands. This is several orders of magnitude

slower than desired (e.g., service-level objectives measured in seconds). This challenge of scaling up resource allocation has prompted a series of recent works [1, 43, 44, 48, 65], but they either rely on domain-specific customizations or workload-dependent assumptions. In this work, we instead seek to take a general (i.e., domain- and workload-agnostic) approach to scaling resource allocation.

Separable problem structure. Upon surveying real-world resource allocation problems from the literature (Table 1), we find that nearly all these formulations can be transformed into a *separable* structure (§1) formally described as follows. (We discuss the generality and limitations in §4.)

- *Variables.* An allocation matrix $x \in \mathcal{X}$ specifies the allocation of n resources among m demands, where x_{ij} indicates the amount or fraction of resource i assigned to demand j (or vice versa). Accordingly, the i-th row of x, denoted as $x_{i*} = (x_{i1}, x_{i2}, \ldots, x_{im})$, constitutes the allocation vector of resource i, while the j-th column $x_{*j} = (x_{1j}, x_{2j}, \ldots, x_{nj})$ encodes the allocation of demand j. The constraint set $\mathcal{X}$ is the Cartesian product of the individual domains $\mathcal{X}_{ij}$, but it is often just non-negative real numbers or integers.
- *Objective.* The objective defines a metric to be maximized or minimized over an allocation matrix x, consisting of a sum of utilities or costs that quantify the outcome from allocating each resource x_{i*} or demand x_{*j}:

$$\sum_i f_i(x_{i*}) + \sum_j g_j(x_{*j}), \tag{1}$$

  where $f_i(\cdot) : \mathbb{R}^m \to \mathbb{R}$ and $g_j(\cdot) : \mathbb{R}^n \to \mathbb{R}$ are utility or cost functions. These functions are typically convex (by nature or by choice), facilitating tractable optimization. A variety of objectives can be transformed into this form using standard optimization techniques. E.g., maximizing the minimum utility can be converted into maximizing an auxiliary "min utility" variable, with additional constraints ensuring that each utility is at least as large as this variable.
- *Constraints.* To enforce feasible allocation, one or more constraints can be imposed on each resource and each demand (e.g., to prevent oversubscription of resources and overprovisioning of demands). For computational tractability, these constraints are typically linear. Following standard optimization practices, we transform inequality constraints into equivalent equality constraints using slack variables. Thus, without loss of generality, we express the $rc_i \geq 1$ resource constraints on each resource i as

$$R_i x_{i*} = r_i, \quad \forall i, \tag{2}$$

  where $R_i \in \mathbb{R}^{rc_i \times m}, r_i \in \mathbb{R}^{rc_i}$ are constraint parameters. Likewise, each demand j is subject to $dc_j \geq 1$ linear constraints:

$$D_j x_{*j} = d_j, \quad \forall j, \tag{3}$$

  where $D_j \in \mathbb{R}^{dc_j \times n}, d_j \in \mathbb{R}^{dc_j}$ are constraint parameters.

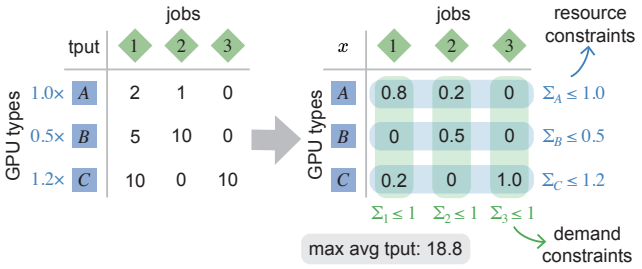


Figure 3: Toy scenario of the job scheduling example.

Despite the separable objective and separate constraints over each resource and each demand, large resource allocation problems (e.g., with millions of variables) still remain painfully slow to solve as they cannot actually be separated into smaller, concurrent subproblems. The main obstacle is that resource and demand constraints are fundamentally entangled. As shown in Equations 2 and 3, each $x_{i,j}$ not only participates in the resource constraints (over x_{i*}) but also in the demand constraints (over x_{*j}).

► **Example:** The example introduced in §1 clearly fits into this formulation. For a more specific example, consider a toy scenario in Figure 3, where three language model inference jobs are scheduled across three types of GPUs. For simplicity, we assume each job fits in a single GPU of any type ($\text{req}_j = 1$), and all jobs have equal priority ($w_j = 1$).

The left panel of Figure 3 shows the capacity of each GPU type (capacity_i) in GPU hours, alongside the throughput table (tput_{ij}). E.g., GPU type C has a total capacity of 1.2 GPU hours, and assigning job 1 to it yields a throughput of 10 TPS.

The right panel shows the optimal allocation matrix (x_{ij}), which satisfies all resource and demand constraints. E.g., the optimal solution schedules job 1 on (one unit of) GPU type A for 0.8 hours, and on GPU type C for 0.2 hours. The maximum average throughput achieved is 18.8 TPS. □

3 DEDE: Decouple and Decompose

We present DEDE, a practical and theoretically rooted optimization framework for solving large-scale, real-world resource allocation problems. By exploiting the inherently separable structure of these problems as described earlier, DEDE achieves massive parallelism while producing near-optimal solutions through a *decouple-and-decompose* approach.

3.1 Decouple

The interdependence between resource and demand constraints is a core complexity in resource allocation. Achieving an optimal solution requires jointly allocating all resources to all demands; otherwise, local, greedy strategies likely result in suboptimal or infeasible solutions.

Existing solutions do not address this intertwined structure of constraints in resource allocation. In practice, production

systems tend to simply rely on commercial solvers and employ linear programming (LP) or mixed-integer linear programming (MILP) algorithms (e.g., simplex methods [46], barrier methods [31], or branch-and-bound methods [7]) to navigate a high-dimensional search space, which is induced by the coupling of resource and demand constraints.

A recent work, POP [44], naively breaks entangled constraints by partitioning resources and demands into a handful of subsets (e.g., 4–64), and then independently allocates a subset of resources to a subset of demands. Doing so ignores the dependencies across subsets and leads to suboptimal solutions in practice (§7). Meanwhile, it only enables limited parallelism because finer-grained decomposition would render the allocation infeasible and unusable.

Instead of disregarding the constraints across subproblems, we tackle the root cause of constraint entanglement. Specifically, without sacrificing solution quality, our high-level goal is to decouple demand constraints from resource constraints, transforming the global allocation into alternating per-resource and per-demand subproblems.

Reformulation. Consider the minimization of the objective in Equation 1 (the signs of f_i and g_j can be simply reversed for maximization problems), subject to resource constraints in Equation 2 and demand constraints in Equation 3.

First, we introduce an auxiliary variable z as a duplicate of the allocation matrix x, and rewrite all demand constraints and all demand-related terms in the objective using z. Meanwhile, we add a new constraint $x = z$. This transformation clearly leads to an equivalent optimization problem:

$$\begin{aligned} \min_{x\in\mathcal{X}} \quad & \sum_i f_i(x_{i*}) + \sum_j g_j(z_{*j}), \\ s.t. \quad & R_i x_{i*} = r_i, \quad \forall i, \\ & D_j z_{*j} = d_j, \quad \forall j, \\ & x - z = 0. \end{aligned} \tag{4}$$

In this formulation, all resource constraints remain on x, while all demand constraints (and related utilities) are moved to z.

Although introducing z appears to separate the problem structurally, the (necessary) constraint $x = z$ still tightly couples the variables and thus prevents straightforward decomposition. In fact, jointly optimizing x and z (e.g., using penalty methods [4] or augmented Lagrangian [23]) will forfeit the benefits of this reformulation, as we will show in §7.3.

ADMM background. Nevertheless, this reformulation lends itself to ADMM (Alternating Direction Method of Multipliers) [8], a classical method for solving constrained optimization problems of the form: minimize $f(x)+g(z)$, subject to $Ax+Bz=c$. Note that Equation 4 conforms to this form.

ADMM is based on augmented Lagrangian [23], a method that converts constrained optimization into a sequence of unconstrained optimization problems using Lagrange multipliers [6] (a well-known technique in optimization). These unconstrained problems are defined over the Lagrangian function, constructed from the objective and constraints: $L_\rho(x,z,y) = f(x)+g(z)+y^\mathsf{T}(Ax+Bz-c)+(\rho/2)\|Ax+Bz-c\|_2^2$, where y is (an estimate of) the Lagrange multiplier, and the last quadratic term with parameter ρ serves as a penalty [4] for improving convexity and stability.

The theory of ADMM enables partial, alternating optimization of the Lagrangian with respect to x and z—iteratively solving for x with z fixed, and solving for z with x fixed. This "two-block partial optimization" guarantees convergence to an optimal solution for convex problems [20, 21]. In non-convex settings (including those involving integers), empirical results indicate that ADMM can still effectively solve many of these problems [54, 64, 70]. Theoretical convergence results are established under certain conditions [39, 57, 61].

By introducing the auxiliary variable $u = (1/\rho)y$, the augmented Lagrangian can be further rewritten as: $L_\rho(x,z,u) = f(x)+g(z)+(\rho/2)\|Ax+Bz-c+u\|_2^2-(\rho/2)\|u\|_2^2$, leading to a more convenient *scaled form* of ADMM. Next, we present the ADMM iterates for updating x, y, and u directly in the context of resource allocation problems.

Applying ADMM. DEDE applies the scaled form of ADMM to solve the reformulated optimization problem (Equation 4). First, it constructs the augmented Lagrangian:

$$\begin{aligned} \mathcal{L}_\rho(x,z,\alpha,\beta,\lambda) &= \sum_i f_i(x_{i*}) + \sum_j g_j(z_{*j}) \\ &+ \frac{\rho}{2}\Big(\sum_i \|R_i x_{i*} - r_i + \alpha_i\|_2^2 + \sum_j \|D_j z_{*j} - d_j + \beta_j\|_2^2 \\ &+ \|x - z + \lambda\|_F^2\Big) - \frac{\rho}{2}(\|\alpha\|_2^2 + \|\beta\|_2^2 + \|\lambda\|_F^2), \end{aligned} \tag{5}$$

where $\alpha_i \in \mathbb{R}^{rc_i}$, $\beta_j \in \mathbb{R}^{dc_j}$, $\lambda \in \mathbb{R}^{n\times m}$ are the introduced auxiliary variables. $\|\cdot\|_2$ still denotes the ℓ^2-norm of a vector, while $\|\cdot\|_F$ is the Frobenius norm of a matrix, i.e., the ℓ^2-norm of the matrix when flattened into a vector.

The ADMM iterates are given by:

$$x^{(k+1)} := \arg\min_{x\in\mathcal{X}} \mathcal{L}_\rho(x, z^{(k)}, \alpha^{(k)}, \beta^{(k)}, \lambda^{(k)}) \tag{6}$$

$$z^{(k+1)} := \arg\min_{z} \mathcal{L}_\rho(x^{(k+1)}, z, \alpha^{(k)}, \beta^{(k)}, \lambda^{(k)}) \tag{7}$$

$$\begin{aligned} \alpha_i^{(k+1)} &:= \alpha_i^{(k)} + R_i x_{i*}^{(k+1)} - r_i, \forall i \\ \beta_j^{(k+1)} &:= \beta_j^{(k)} + D_j z_{*j}^{(k+1)} - d_j, \forall j \\ \lambda^{(k+1)} &:= \lambda^{(k)} + x^{(k+1)} - z^{(k+1)} \end{aligned}$$

We will show in §3.2 that both the optimization subproblems in Equations 6 and 7 are amenable to decomposition owing to the previously identified separable problem structure.

3.2 Decompose

Up to this point, we have reformulated the original resource allocation problem into alternating minimization steps of the

augmented Lagrangian with respect to x and z. Next, we take the x-minimization step as an example to describe how the optimization is decomposed (z-minimization is similar).

Because the x-minimization step solves for x only, we can ignore all terms in the augmented Lagrangian without x (such as demand constraints). Thus, we rewrite Equation 6 as

$$\begin{aligned} x^{(k+1)} &= \arg\min_{x\in\mathcal{X}} \mathcal{L}_\rho(x, z^{(k)}, \alpha^{(k)}, \beta^{(k)}, \lambda^{(k)}) \\ &= \arg\min_{x\in\mathcal{X}} \sum_i f_i(x_{i*}) + \frac{\rho}{2}\sum_i \left\|R_i x_{i*} - r_i + \alpha_i^{(k)}\right\|_2^2 \\ &\quad + \frac{\rho}{2}\left\|x - z^{(k)} + \lambda^{(k)}\right\|_F^2 \\ &= \arg\min_{x\in\mathcal{X}} \sum_i \Big(f_i(x_{i*}) + \frac{\rho}{2}\left\|R_i x_{i*} - r_i + \alpha_i^{(k)}\right\|_2^2 \\ &\quad + \frac{\rho}{2}\left\|x_{i*} - z_{i*}^{(k)} + \lambda_{i*}^{(k)}\right\|_2^2\Big). \end{aligned} \tag{6}$$

Given that the new objective is a sum of disjoint terms across index i, its minimization can be decomposed into independent, per-resource subproblems over each x_{i*}:

$$\begin{aligned} x_{i*}^{(k+1)} = \arg\min_{x_{i*}\in\mathcal{X}_{i*}} \Big(&f_i(x_{i*}) + \frac{\rho}{2}\left\|R_i x_{i*} - r_i + \alpha_i^{(k)}\right\|_2^2 \\ &+ \frac{\rho}{2}\left\|x_{i*} - z_{i*}^{(k)} + \lambda_{i*}^{(k)}\right\|_2^2\Big), \quad \forall i. \end{aligned} \tag{8}$$

This decomposition is made possible by the separable structure: (1) the original objective is separable, with a utility $f_i(x_{i*})$ defined for allocating each resource i, and (2) the resource constraints are likewise separable, as each constraint $R_i x_{i*} = r_i$ pertains only to the corresponding resource i.

As a result, the original optimization problem over $n \times m$ variables is broken down into n independent subproblems, each involving only m variables. DEDE solves each subproblem using off-the-shelf solvers, which return either exact or approximate solutions depending on the nature of f (convex or not), whether $\mathcal{X}$ contains integers, etc. Since this decomposition allows up to n parallel computations—often numbering in the thousands in large-scale settings—DEDE is capable of delivering substantial speedups assuming sufficient CPUs.

For linear programs, the original problem has a worst-case time complexity of approximately $O((n\cdot m)^{2.373})$ [13, 44]. After decomposition, DEDE processes n subproblems, each with a complexity of $O(m^{2.373})$, reducing the overall complexity to $O(n\cdot m^{2.373})$. Nonetheless, this theoretical improvement assumes that ADMM converges within a constant number of iterations. In practice, the number of iterations depends on the choice of the penalty parameter ρ (which acts like a "learning rate") and other factors that influence convergence.

Similarly, the z-minimization step decomposes into independent, per-demand allocation subproblems over each z_{*j}:

$$\begin{aligned} z_{*j}^{(k+1)} = \arg\min_{z_{*j}} \Big(&g_j(z_{*j}) + \frac{\rho}{2}\left\|D_j z_{*j} - d_j + \beta_j^{(k)}\right\|_2^2 \\ &+ \frac{\rho}{2}\left\|x_{*j}^{(k+1)} - z_{*j} + \lambda_{*j}^{(k)}\right\|_2^2\Big), \quad \forall j. \end{aligned} \tag{9}$$

4 Generality and Limitations

In this section, we discuss the generality of DEDE (conditions under which it applies) and its limitations (scenarios that may result in limited parallelism or suboptimal solutions).

4.1 Generality

DEDE is applicable to a wide range of real-world resource allocation problems. Table 1 summarizes these problems that appear in recent literature such as OSDI, SOSP, NSDI, and SIGCOMM. Below, we explain why DEDE is applicable.

- *Variables.* As an ADMM-based framework, DEDE natively supports continuous (real-valued) allocation variables. Empirical evidence [54] and theoretical proofs [39, 57, 61] on ADMM further indicate that DEDE should effectively handle boolean and integer variables by projecting real-valued solutions onto the appropriate domains during iterations.
- *Objective.* Most objectives in real systems are defined as the sum, max, or min of utility functions over the individual allocation of resources and demands. They also exhibit convexity. Examples include weighted sums [45, 65], logarithmic utilities [2], and quadratic costs [60, 66].
- *Constraints.* DEDE assumes linear constraints on resources and demands—a condition that holds across all the surveyed use cases. These constraints are commonly expressed at the per-resource or per-demand level. Even for constraints that do span multiple resources or demands, DEDE can turn them into disjoint constraint groups and apply decomposition at the group level (at the cost of reduced parallelism).

Unlike POP [44], DEDE works with any workload, not only "granular" ones. In POP, the demands in each subproblem can only access a subset of resources, assuming each demand only requests a small fraction of interchangeable resources such that a resource subset suffices. This granular assumption can be unrealistic in practice, due to uneven demand distributions [29, 58] or specific resources that each demand is allowed to utilize [34, 59]. In contrast, each subproblem in DEDE can still access all the resources or demands, resulting in generalization to various workload distributions.

4.2 Limitations

While DEDE applies to a variety of resource allocation tasks, it might not always be the optimal approach for all scenarios.

Limited parallelism. DEDE's parallelism relies on its ability to decompose the problem into independent subproblems. When such decomposition is not fully achievable, the degree of parallelism diminishes. Although this limitation does not manifest in the set of real-world problems we surveyed, we come up with some examples below.

- *Non-separable objectives.* An objective is non-separable when utility depends on the interaction between different

	Variables			Objective	
	Boolean	Integer	Float	Linear	Convex
RDC [56]	✓			✓	
SkyPilot [66]	✓				✓
ARROW [69], FlexWAN [40]	✓	✓		✓	
Shoofly [52]		✓	✓	✓	
PODP [5], RAS [47], Skyplane [27], Oort [35], TACCL [50], Shard Manager [38], Zeta [67], CASCARA [51], Sia [28], POP [44]	✓		✓	✓	
NetHint [12], Gavel [45], Teal [65], ONEWAN [33], BLASTSHIELD [32], NCFlow [1], Cerebro [68], DOTE [48], POP [44]			✓	✓	
PCF [30], Electricity Pricing [60], POP [44]			✓		✓

Table 1: Real-world resource allocation problems in recent literature.

resources or demands, e.g., if a job attains higher throughput when it has exclusive access to a GPU compared with when sharing it with other jobs.

- *Non-separable constraints.* A constraint is non-separable when it spans multiple resources and multiple demands simultaneously. For example, this occurs when the jobs submitted by each user are collectively limited by a user-specific GPU hour quota. To restore decomposability, DeDe has to treat all jobs (the original demands) from the same user as a single aggregated demand. However, this aggregation reduces the granularity of parallelism, from the job level to the user level.

Suboptimal solutions. DeDe may fail to reach the optimal solution when the allocation problem deviates from the assumptions underlying standard ADMM.

- *Non-convex problems.* Although ADMM has demonstrated empirical success on various non-convex problems [54, 64, 70], its theoretical convergence is guaranteed only under specific conditions [39, 57, 61]. Consequently, DeDe may converge to suboptimal solutions in non-convex settings.
- *Nonlinear constraints.* When a constraint is nonlinear, DeDe has to reformulate it with an indicator function in the augmented Lagrangian. The indicator function returns zero if the constraint is satisfied and infinity otherwise. However, this approach destroys the convexity of the objective, resulting in the aforementioned non-convex problems.
- *Higher allocation dimension.* DeDe assumes a two-dimensional allocation matrix that maps resources to demands, leading to a two-block ADMM formulation. Introducing additional dimensions such as a temporal dimension [37] will increase the number of variable blocks in ADMM and thus might lead to convergence issues [11].

5 Case Studies

In this section, we present case studies of DeDe applied to three representative resource allocation problems [44]: cluster scheduling, traffic engineering, and load balancing.

5.1 Cluster Scheduling

Cluster scheduling problems [43–45, 63] involve allocating computing jobs to different resource types in heterogeneous clusters. The job scheduling example introduced earlier is a simplified version of this problem.

Max-min allocation. This variant aims to distribute the heterogeneous cluster resources fairly among jobs such that the minimum throughput is maximized.

- *Variables.* Jobs can be time-sliced across available resource types [44, 45]. Thus, the allocation plan is a matrix $x \in [0,1]^{n\times m}$, where x_{ij} is the fraction of time (optimization interval) that each job j spends on each resource type i.
- *Objective.* Given an allocation matrix x, we adopt a normalized effective throughput of job j from POP [42], which is a linear function of x_{*j}. The objective is to maximize the minimum normalized effective throughput:

$$\underset{x}{\text{Maximize}} \ \min_{j} \ \text{throughput}(j, x_{*j}).$$

- *Resource constraints.* For each computing resource type i, the total amount of resources requested by all jobs cannot exceed the corresponding capacity:

$$\sum_j x_{ij} z_j \le \text{capacity}_i, \quad \forall i,$$

where z_j is the amount of resources requested by job j.

- *Demand constraints.* For each job j, the sum of all fractions of time must not exceed 1 naturally:

$$\sum_i x_{ij} \le 1, \quad \forall j.$$

Proportional fairness. The second variant of cluster scheduling maximizes the overall resource utilization while ensuring minimum service for each job. This variant shares the same variables and constraints as above, except for the objective:

- *Objective.* Proportional fairness aims to maximize the sum of log utilities of each job:

$$\underset{x}{\text{Maximize}} \sum_j \log(\text{throughput}(j, x_{*j})).$$

Applying DEDE. Both variants of cluster scheduling exhibit the separable structure required by DEDE. Their objectives to maximize are a sum or min of utilities for allocating each job, and a linear constraint is applied to each job and each computing resource. Thus, DEDE can decompose the problem into n per-job allocations and m per-demand allocations.

5.2 Traffic Engineering

Traffic engineering problems [24,43,44,55,65] allocate traffic demands between datacenters onto the network links in a wide-area network (WAN) topology.

Maximizing total flow. This problem variant focuses on maximizing the total flow allocation.

- *Variables.* Let $x_{(u,v)(s,t)} \geq 0$ denote the amount of flow from a datacenter pair (s,t) assigned to a network link (u,v). In common path-based traffic engineering, flows between each node pair (s,t) are allocated only over links along pre-configured paths $P_{(s,t)}$.
- *Objective.* For each node pair (s,t), the inflow it incurs to node v is computed as: $\text{inflow}(v,(s,t)) = \sum_{(u,v)\in P_{s,t}} x_{(u,v)(s,t)}$, and outflow as: $\text{outflow}(v,(s,t)) = \sum_{(v,u)\in P_{s,t}} x_{(v,u)(s,t)}$. Consequently, the total flow allocated for the node pair (s,t) is captured by $\text{inflow}(t,(s,t))$, the total inflow toward the destination t. Therefore, the objective of maximizing the total flow between all node pairs can be expressed as:

$$\underset{x}{\text{Maximize}} \sum_{(s,t)} \text{inflow}(t,(s,t)).$$

- *Resource constraints.* For each link (u,v), the cumulative flow from all node pairs traversing it must not exceed its link capacity $c_{u,v}$:

$$\sum_{(s,t)} x_{(u,v)(s,t)} \leq c_{u,v}, \quad \forall (u,v) \in E.$$

- *Demand constraints.* For each node pair (s,t), the total flow allocated cannot exceed the total demand. Additionally, for all nodes other than s and t, the inflow and outflow must be equal, ensuring no flow "black hole" along the paths:

$$\begin{aligned} &0 \leq \text{inflow}(t,(s,t)) \leq d_{s,t}, \\ &\text{inflow}(v,(s,t)) = \text{outflow}(v,(s,t)), \forall v \neq s,t, \end{aligned} \quad \forall (s,t).$$

Minimizing max link utilization. This variant aims to evenly distribute traffic demands across network links to avoid overloading any single link. The utilization of link is commonly defined as the (uncapped) ratio of total flow assigned to a link to the link's capacity, although it cannot actually exceed 100% in practice. This formulation shares the same variables and constraints as above, with a revised objective:

- *Objective.* The goal is to minimize the max link utilization across the network:

$$\underset{x}{\text{Minimize}} \max_{(u,v)\in E} \frac{\sum_{(s,t)} x_{(u,v)(s,t)}}{c_{u,v}}.$$

Applying DEDE. The traffic engineering problem aligns with the separable structure of DEDE: the objectives are expressed as either a sum over per-demand utilities or as a maximum over per-link utilities, and linear constraints are applied to each demand and each link. Therefore, DEDE can decompose the problem into $|E|$ per-link allocations and $|V|^2$ per-demand allocations for parallelism. In practice, due to the large overhead of maintaining $|V|^2$ per-demand subproblems, we group these subproblems by their sources, reducing the total number of subproblems to just $|V|$.

5.3 Load Balancing

Load balancing problems [14,44,49,53] allocate data shards among storage servers to scale out query loads in a distributed store. Load balancing is a non-convex problem with integer variables, wherein DEDE can provide a high-quality solution empirically, as discussed in §4.2.

Minimizing shard movements. This problem aims to minimize shard movements across servers during load changes while keeping the load nearly balanced on each server.

- *Variables.* The allocation plan is encoded in a matrix $x \in [0,1]^{n\times m}$, where x_{ij} is the fraction of data shard j assigned to server i. Additionally, a binary matrix x' captures shard placement, where $x'_{ij} = 1$ if the data shard j is located on server i (i.e., when $x_{ij} > 0$); otherwise, $x'_{ij} = 0$.
- *Objective.* The movement of data shard j allocated to server i can be represent as $(1 - T_{ij})x'_{ij}f_i$, where T_{ij} denotes the initial shard placement, and f_j denotes the memory footprint of data shard j. The objective of minimizing total shard movement can be expressed as:

$$\underset{x}{\text{Minimize}} \sum_j \sum_i (1 - T_{ij})x'_{ij}f_i.$$

- *Resource constraints.* For each server i, the total query load must be close to the average query load L. Meanwhile, the total memory usage must be within the memory capacity:

$$\begin{aligned} &L - \varepsilon \leq \textstyle\sum_j x_{ij}l_j \leq L + \varepsilon, \\ &\textstyle\sum_j x'_{ij}f_j \leq \text{memory}_i, \end{aligned} \quad \forall i,$$

  where l_j is the query load on data shard j.

- *Demand constraints.* Each data shard j must be entirely allocated across servers:

$$\sum_i x_{ij} = 1, \quad \forall j.$$

Applying DEDE. The load balancing problem also demonstrates a separable structure: the objective is the sum of allocation quality per data shard, while linear constraints are applied independently to each server and each shard. This allows DEDE to decompose the problem into n per-server allocations and m per-shard allocations.

6 Implementation of DEDE

We have implemented DEDE as a Python package and uploaded it to PyPI for easy installation: `pip install dede`.

Listing 1 showcases a basic resource allocation example using DEDE. Line 5 creates an $N \times M$ non-negative allocation matrix `x`, where `x[i,j]` represents the fraction of time that demand `j` runs on resource `i`. Lines 12–13 define a constraint for each resource using parameters, and Lines 14–15 define a constraint for each demand. The objective, defined in Line 18, is to maximize the sum of entries in `x`. Finally, the problem is constructed in Lines 21–22 and solved in Line 23.

Built on top of the popular optimization modeling language `cvxpy` [3, 15], DEDE inherits most of its syntax and APIs, such as `Variable()`, `Parameter()`, and `Maximize()`. A notable distinction is that DEDE requires users to explicitly separate resource constraints and demands constraints when initializing a problem (Line 22). As a parallel optimization tool, DEDE allows users to configure the number of CPUs (Line 23); by default, all available cores are used.

Internally, DEDE solves the optimization in three stages:

- *Problem parsing.* DEDE transforms all inequality constraints into equivalent equality constraints through non-negative slack variables. E.g., `x[:,j].sum() <= 1` is converted to `x[:,j].sum() + sj == 1` with a slack variable `sj`. Then slack variables are treated just like other variables.
- *Problem building.* DEDE organizes resource constraints into disjoint per-resource groups and demand constraints into disjoint per-demand groups. For each group, a per-resource or per-demand subproblem is constructed with `cvxpy` following Equations 8 and 9.
- *Problem solving.* DEDE solves these subproblems in parallel using multiple CPU cores. During ADMM iterations, only the parameters are updated, avoiding the overhead of rebuilding problems in `cvxpy`. Similarly, for the same problem with varying resources and demands, only the relevant parameters are updated.

We use Ray [41] to parallelize our computation across multiple cores, as Python's native multithreading model is constrained by the (notorious) global interpreter lock (GIL). Ray circumvents this issue by managing multiple Python interpreter processes, effectively bypassing the GIL. Moreover, Ray offers robust support for inter-process communication, sparing us from manually implementing such mechanisms using Python's lower-level multiprocessing primitives.

```
1  import numpy as np
2  import dede as dd
3
4  # Create allocation variables
5  x = dd.Variable((N, M), nonneg=True)
6
7  # Create parameters
8  param = dd.Parameter(
9      N, value=np.random.uniform(0, 1, N))
10
11 # Create constraints
12 resource_constrs = [
13     x[i,:].sum() <= param[i] for i in range(N)]
14 demand_constrs = [
15     x[:,j].sum() <= 1 for j in range(M)]
16
17 # Create an objective
18 obj = dd.Maximize(x.sum())
19
20 # Construct and solve the problem
21 prob = dd.Problem(
22     obj, resource_constrs, demand_constrs)
23 prob.solve(num_cpus=64, solver=dd.ECOS)
```

Listing 1: Resource allocation example with DEDE package.

7 Evaluation

We answer the following questions in the evaluation:

1. How does DEDE compare with state-of-the-art approaches in allocation quality and computation time? (§7.1)
2. How do DEDE and baseline approaches react to changes in problem granularity, temporal fluctuation, spatial redistribution, and link failures? (§7.2)
3. What is the individual contribution of each design component in DEDE? (§7.3)

Our evaluation compares four approaches:

- *Exact sol.* This baseline solves the original resource allocation problem using commercial solvers. We adopt the implementation from POP, which utilizes `cvxpy` [3, 15], Gurobi [22], and CPLEX [25] for cluster engineering, traffic engineering, and load balancing, respectively.
- *POP.* POP-k [44] randomly splits the problem into k smaller subproblems, applies commercial solvers to each, and coalesces the resulting k sub-allocations into a global allocation. However, POP only *simulates* the parallel execution by solving subproblems sequentially and calculating the parallel solving time mathematically.
- DEDE. DEDE is our truly parallel implementation (§6). By default, the solution from the previous optimization interval is used to warm-start the subsequent interval.
- DEDE*. For a fair comparison with POP, we create a variant of DEDE, denoted as DEDE*, following POP's simulation methodology. DEDE* solves subproblems sequentially and estimates the parallel solving time mathematically.

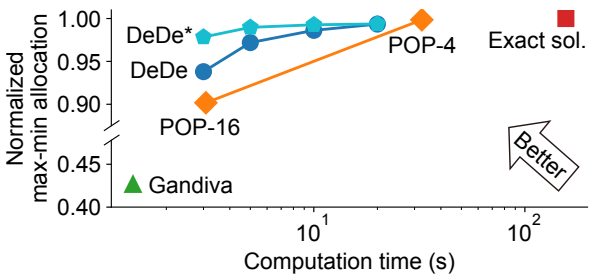


Figure 4: Results of the cluster scheduling variant that maximizes the minimum job throughput (max-min allocation).

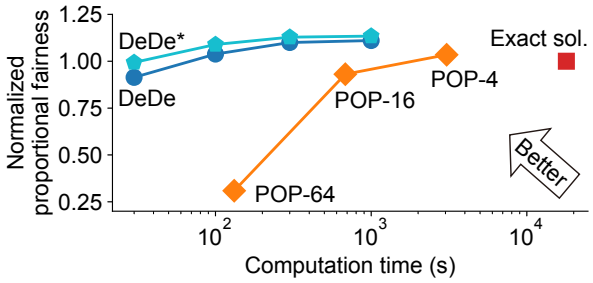


Figure 5: Results of the cluster scheduling variant that maximizes proportional fairness among job throughputs.

We also consider domain-specific approaches apart from these four approaches. In cluster scheduling and load balancing, we evaluate the greedy heuristic algorithms, Gandiva [63] and E-Store [53], respectively. In traffic engineering, we evaluate a demand-pinning approach ("Pinning") similar to a prior work [42], where the top 10% of demands are allocated using optimization engines and the rest are assigned to shortest paths. We also evaluate Teal [65], a learning-accelerated traffic engineering approach that achieves massive parallelism using a GPU.

All approaches are evaluated in terms of allocation quality and computation time. The computation time is measured using 64 CPU cores (2× Intel Xeon Gold 6142), with an additional GPU (Nvidia Titan RTX) available for Teal.

7.1 DeDe vs. the State of the Art

In this section, we compare DeDe against state-of-the-art approaches on resource allocation problems from §5.

7.1.1 Cluster Scheduling

We stress-test DeDe's scalability in a large, heterogeneous cluster scheduling environment, where jobs are allocated in a time-sliced manner across 456 different types of compute resources, e.g., GPU/CPU instances that vary in vendor, generation, memory capacity, and other specifications. The quantity of each resource type is randomly drawn from $\{8, 16, 24, \ldots, 64\}$, resulting in a total of 16,520 instances. We utilize the simulator in Gavel [45] and model job arrivals as a Poisson process with an average inter-arrival of 100 seconds. We update job allocation decisions every 6 minutes, repeating this process for 200 scheduling rounds (20 hours). For each job, the number of requested resource instances per type is drawn from $\{1, 2, 4, 8, 16, 32\}$, and following prior work [59], 33% of the jobs are restricted to specific resource types. Job throughput is derived from relevant benchmarks [19, 26, 36]. Please refer to Appendix A for more details on this setup.

Max-min allocation. Figure 4 compares DeDe against other approaches in terms of max-min allocation. DeDe quickly attains a near-optimal max-min allocation of 0.94 in just 3 seconds and 0.99 within 10 seconds, demonstrating a better balance between max-min allocation and computation time compared with Exact sol., Gandiva, and POP. In contrast, Exact sol. experiences a significantly slower computation time of 156 seconds due to its sequential nature. The greedy heuristic, Gandiva, performs poorly, achieving only a max-min allocation of 0.43 (in 1.4 seconds). POP-4 provides a comparable tradeoff between time and accuracy to DeDe, but is 1.6× slower. The faster variant, POP-16, splits the problem into 16 smaller subproblems and achieves a lower max-min allocation of 0.9 (in 3.1 seconds), since smaller subproblems limit the ability to select optimal GPU/CPU assignments.

DeDe*, representing the same algorithm under idealized conditions similar to POP's simulation, achieves a normalized max-min allocation of 0.99 and is 3.3× faster than DeDe in Figure 4. The reason is that DeDe* executes each iteration more quickly and completes more iterations within a given time. The discrepancy in iteration time between DeDe and DeDe* arises from three factors: (1) DeDe measures the end-to-end time, including problem solving, compilation, solution unpacking, and other associated tasks, while DeDe* only accounts for the core solving time; (2) DeDe's parallel implementation incurs overhead from cache contention, which slows computation; (3) In DeDe, each subproblem is statically pre-assigned to one of the processes, making it susceptible to straggler delays. In contrast, DeDe* assumes perfect dynamic scheduling.

Proportional fairness. Figure 5 compares DeDe against other approaches in maximizing proportional fairness. This problem is particularly challenging due to the logarithmic form of the objective function. Exact sol., which uses the SCS solver in `cvxpy`, fails to reach optimality even after 5 hours of computation. As a result, several methods report normalized fairness scores that exceed 1 when normalized relative to Exact sol. Both DeDe and DeDe* achieve greater speedups than POP by effectively breaking the problem into smaller and easier subproblems. Notably, DeDe and DeDe* both reach a normalized proportional fairness of over 1 within 100 seconds. In contrast, POP-4 and POP-16 require 3,053 seconds and 682 seconds, respectively, to achieve comparable allocation quality, while POP-64 yields a significantly lower

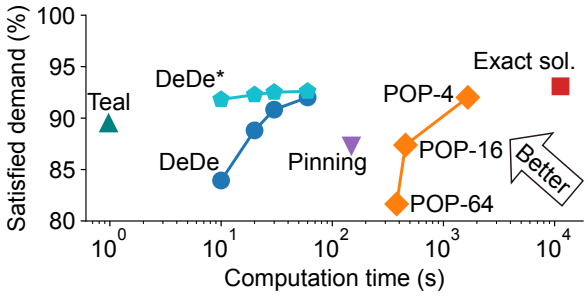


Figure 6: Results of the traffic engineering problem variant that maximizes the total flow.

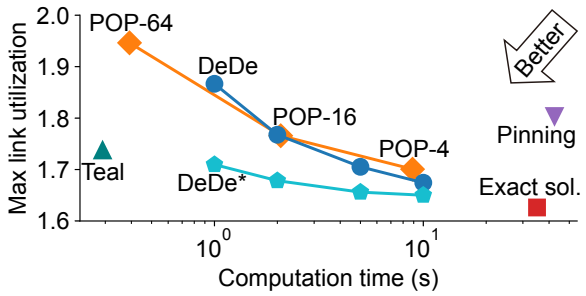


Figure 7: Results of the traffic engineering problem variant that minimizes the max link utilization.

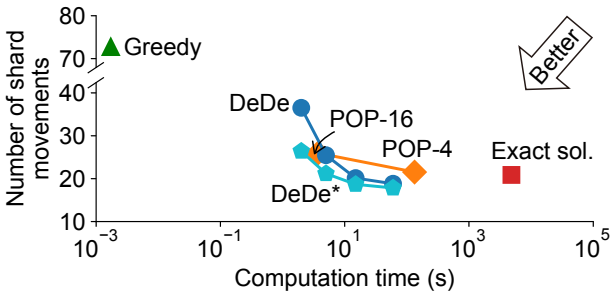


Figure 8: Results of the load balancing problem where the objective is to minimize shard movements.

fairness score of only 0.31.

7.1.2 Traffic Engineering

We adopt the evaluation setup in Teal [65], with a focus on its largest topology—a 1,739-node network adapted from an internet topology for WAN purposes. The traffic matrices are collected from the production WAN of a global cloud provider and mapped onto the above topology.

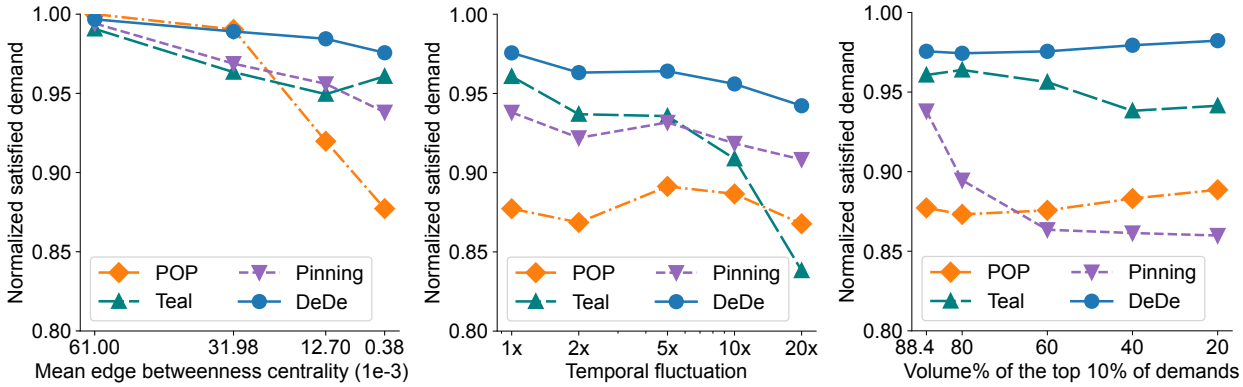
Maximize total flow. Figure 6 compares DEDE to the baselines in maximizing total flow. DEDE achieves 90.8% satisfied demand within 30 seconds and 92% within 60 seconds, showing a superior trade-off between satisfied demand and computation time compared with Exact sol., POP, and Pinning. Its idealized variant, DEDE*, achieves a satisfied demand that is 8% higher than DEDE within 10 seconds. While POP-4 also reaches 92% satisfied demand, its average computation time is significantly longer, at 1,658 seconds. POP-16 and POP-64 improve parallelism by dividing the workload into more subproblems, reducing computation times to 456 seconds and 380 seconds, respectively. Note that the speedup of POP-64 is limited compared with POP-16 as the number of cores per subproblem drops from 4 to 1. This increased parallelism also comes at a cost, with satisfied demand decreasing to 87.4% for POP-16 and 81.6% for POP-64. Pinning focuses on top demands to reduce problem size but remains constrained by the sequential nature of its optimization solvers. As a result, it achieves a modest speedup, requiring 149 seconds to reach 87.3% satisfied demand. Teal, a learning-accelerated approach, leverages massive GPU parallelism to achieve 89% satisfied demand in only 1 second. However, it relies on a highly customized machine-learning framework, demanding significant human effort for design and training.

Min-max link utilization. Figure 7 compares DEDE with other approaches in minimizing maximum link utilization. Here, the link utilization metric serves as proxy for network congestion and is allowed to exceed 100% during optimization. DEDE attains a maximum link utilization of 1.67 within 10 seconds and 1.71 within 5 seconds. Exact sol. reaches 1.63 with a longer computation time of 35 seconds. This is faster than maximizing total flow because minimizing max link utilization is computationally less challenging and thus requires fewer iterations. Pinning takes 42.5 seconds, making it slower than DEDE and slightly slower than Exact sol., as it requires additional time to rebuild the problem when the top 10% of node pairs change. POP's variants, POP-4, POP-16, and POP-64, achieve a maximum link utilization of 1.70, 1.77, and 1.95 within 9, 2, and 0.4 seconds on average, respectively. Although POP-16 is faster than DEDE, a fairer comparison between POP-16 and DEDE*, both simulating parallelism, reveals that DEDE* achieves a comparable computation time to POP-16. The domain-customized Teal takes only 0.3 seconds on a GPU to attain a maximum link utilization of 1.74.

7.1.3 Load Balancing

We follow the load balancing settings in POP [44], except for increasing the scale to 2,048 data shards distributed across 256 servers. In each round, a new shard-to-server allocation is computed based on updated query loads. We simulate for 100 rounds, using the first 20 as a warm-up. To ensure the problem's feasibility, we set the tolerance parameter ϵ to 0.1.

Minimize shard movements. Figure 8 compares DEDE with baselines in minimizing shard movements. DEDE achieves an average of 20.1 shard movements in 15 seconds, better than POP-4's 21.5 shard movements within 133 seconds. In faster scenarios, DEDE achieves 25.4 shard movements in 5 seconds, while POP-16 achieves an average movement of 26 in a slighter less time of 3.7 seconds. For a fair comparison,



(a) Granularity changes. (b) Temporal fluctuation. (c) Spatial redistribution.

Figure 9: Normalized satisfied demand in traffic engineering against granularity, temporal, and spatial changes.

DeDe* consistently achieves around 16% fewer shard movements than POP, given a similar time limit. Exact sol. requires 20.9 shard movements on average, but is significantly slower with an average of 4,820 seconds. This slower computation is due to the NP-hard nature of solving the mixed-integer linear program underlying load balancing. Notably, Exact sol. does not produce the minimum possible shard movements because optimizing each scheduling round independently does not guarantee an optimal solution across the entire series. At the opposite end of the time-accuracy trade-off, the greedy heuristic provides a rapid but less accurate solution, completing in just 2 milliseconds but averaging 73 shard movements after naively fixing its constraint violations.

7.2 Robustness

Using traffic engineering as an example, we evaluate the robustness of DeDe and other baselines against changes in problem granularity, temporal dynamics, spatial redistribution, and link failures. In these experiments, DeDe is run for 30 seconds, and we normalize the satisfied traffic demand by the optimal one achieved by Exact sol.

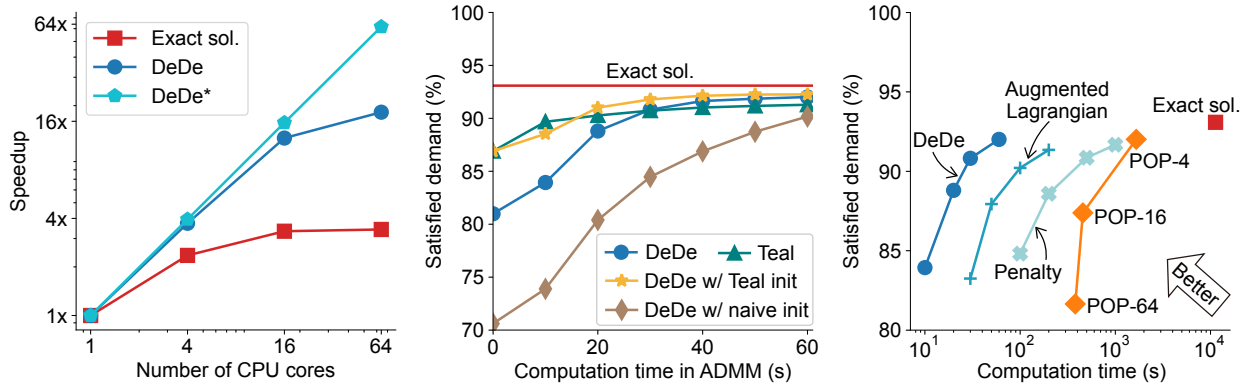
Changes of problem granularity. The granular problem structure required by POP, where each demand only requests a small fraction of *interchangeable* resources, can be impractical. In traffic engineering, for example, demands are constrained by the link capacity on specific pre-configured paths. Similarly, in cluster scheduling, 33% of GPU tasks in production clusters are limited to specific GPU types [59]. In Figure 9a, we present results on varying levels of problem granularity using traffic engineering as an example. To quantify resource interchangeability, we use the mean edge betweenness centrality [9, 10], which measures the average percentage of demands served by a given edge.

As the mean edge betweenness centrality decreases, all approaches experience a decline in normalized satisfied demand, ranging from 0.8–2.1% for DeDe, 2.8–4.2% for Teal, 2.5–5.6% for Pinning, and 1–12.3% for POP. However, the maximum decrease in POP is 5.9× greater than DeDe because POP's subproblems struggle to find alternative resources within their restricted subsets, whereas DeDe's subproblems retain access to the full set of resources. Teal exhibits a slight 1.3% increase in satisfied demand when the mean edge betweenness centrality is minimal. This is because the reduced edge selection simplifies allocation decisions.

Temporal dynamics. Next, we introduce temporal fluctuations [65] to the traffic matrices for robustness assessment. For each demand, we calculate the variance σ^2 in its changes between consecutive time slots and create a new normal distribution $N(0, k\sigma^2)$. We choose the values of k to be 2, 5, 10, and 20. Next, we randomly draw a sample from the newly created normal distribution and add it to each demand in every time slot, simulating temporal fluctuations.

Figure 9b demonstrates that DeDe effectively manages temporal fluctuations, with changes in normalized satisfied demand limited to 1.3–3.5%. In contrast, Teal suffers a significant decrease of 12.8%, as it lacks prior exposure to the distribution of these fluctuating demands during training. Pinning and POP experience variations in normalized satisfied demand ranging from 0.6–3% and 1–1.6%, respectively. This is because randomness may occasionally favor splitting a problem into subproblems or focusing on top demands exclusively, while at other times, it does not.

Spatial redistribution. To evaluate robustness against spatial redistribution, we reassign traffic demands among node pairs. Specifically, we adjust the top 10% of demands, which initially account for 88.4% of the total volume, so that they take up 80%, 60%, 40%, and 20% of the total volume instead. Figure 9c shows that DeDe consistently fulfills the highest demand across all spatial distributions. Pinning's performance decreases by 8.3%, as its strategy of focusing only on top demands relies on the presence of a heavy-tailed demand distribution. Teal has a slight decrease of up to 2.4% due to unseen demand distribution in training. This is a relatively



(a) Speedup when varying CPU cores. (b) Convergence rate. (c) Comparing different optimization methods.

Figure 10: Micro-benchmarks with the objective of maximizing total flow in traffic engineering.

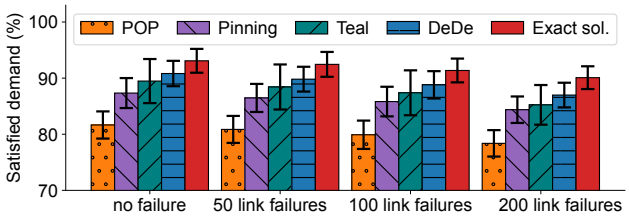


Figure 11: Satisfied demand of traffic engineering under zero, 50, 100, or 200 link failures after recomputing flow allocation.

smaller decrease than the temporal fluctuation, because Teal can still handle simple, evenly distributed traffic demands. POP has a slight increase of 1.4% as demands are more granular, where no demand requires a significant fraction of the total resources.

Link failures. Widespread failures across inter-datacenter links are relatively rare in real-world operations unless triggered by physical fiber cuts [52]. However, we simulate severe failure scenarios, introducing 50, 100, and 200 simultaneous link failures. As shown in Figure 11, the satisfied demand declines consistently across all methods. Specifically, all approaches drop by 0.6–1.1% under 50 link failures, 1.5–2.1% under 100 link failures, and 3–4.2% under 200 link failures. These drops stem from the fact that the failed links represent only a small portion of the total 8,558 links, allowing all approaches to recompute flow allocation given adequate time.

7.3 Micro-benchmarks

We conduct micro-benchmarks to evaluate the impact of DEDE's design components, with all experiments performed in maximizing total flow in traffic engineering.

Speedup when varying CPU cores. Figure 10a evaluates the speedup of DEDE, DEDE*, and Exact sol. with 1, 4, 16, and 64 CPU cores. DEDE* achieves 61.7× speedup under 64 cores, where the slight gap occurs as we cannot exactly divide the total computation time evenly across 64 cores. DEDE also achieves an almost linear speedup when the number of CPU cores grows from 1 to 16. However, this linearity diminishes with 64 cores, where DEDE only achieves 18.2× speedup. This is because utilizing all 64 cores for parallelism incurs overhead from cache contention and stragglers, as discussed in §7.1.1. In contrast to the linear speedup of DEDE* and DEDE, the speedup of Exact sol. is sublinear and marginal, achieving only 3.4× speeds with 64 CPU cores. This is because of commercial solvers' sequential nature, e.g., the simplex method [46] in linear programming requires thousands to millions of small steps to reach the optimal solution.

Convergence rate. Teal generates a coarse, initial solution to traffic engineering using neural networks and fine-tunes it through ADMM. While Teal also "decouples" constraints, it does not further "decompose" the problem as DEDE does. Instead, it analytically and approximately minimizes a problem-specific augmented Lagrangian, directly projecting the result onto the non-negative feasible region. As a specialized method, Teal converges rapidly, achieving 90% satisfied demand within 10 seconds, as shown in Figure 10b. In comparison, DEDE requires ∼30 seconds to attain comparable performance. In each optimization interval, DEDE is warm-started using the previous solution by default. When initialized instead with Teal's neural network output, DEDE converges at a similar rate. In contrast, employing a naive initialization—where each demand is equally split across available paths—reduces DEDE's convergence speed by nearly half.

Alternative optimization methods. Figure 10c evaluates alternative constrained optimization techniques for maximizing total flow in traffic engineering—the penalty method [4] and the augmented Lagrangian method [23], comparing them with the ADMM-based approach used by DEDE. Both methods aim to solve the reformulated optimization problem in Equation 4 by jointly (and naively) optimizing variables x and z, but they differ in how they enforce constraints. The penalty method introduces penalty terms into the objective function to discourage constraint violations. It starts with a

small penalty coefficient and gradually increases it toward infinity, thereby driving the solution toward feasibility. However, this process typically involves solving a series of increasingly ill-conditioned problems, resulting in slow convergence. In our evaluation, it was more than 30× slower than DeDe in achieving a satisfied demand of over 90%. The augmented Lagrangian method improves upon the penalty method by combining both penalty terms and Lagrange multipliers. This helps enhance convergence stability and typically reduces the number of required iterations. However, unlike the ADMM approach used in DeDe, it lacks the benefit of decomposition and parallelism. As a result, despite its faster convergence over the basic penalty method, it remains over 3× slower than DeDe in reaching a satisfied demand of over 90%.

8 Related Work

Resource allocation problems. Multi-tenant systems often address resource allocation challenges. Section 5 examines three prominent examples: cluster scheduling [28, 45, 47, 63, 66] allocates jobs to CPUs or GPUs in heterogeneous clusters, traffic engineering [1, 24, 32, 51, 69] allocates traffic demands to network link capacities in wide-area networks, and load balancing [14, 49, 53] allocates query loads to servers in distributed databases. Beyond these problems, Skyplane [27] allocates cloud resources (e.g., bandwidth and TCP connections) to overlay paths that traverse intermediate cloud regions, minimizing the cost of inter-region bulk transfers. Zeta [67] allocates traffic forwarding capability from gateway clusters to virtual private clouds for scalable and robust east-west communication in large-scale clouds. Shoofly [52] allocates light wavelengths of optical fiber to network shortcuts abstracted from optical bypasses, in order to minimize the hardware costs of provisioning long-haul capacity. By leveraging the commonly observed separable structure of real-world resource allocation problems, DeDe is applicable to the aforementioned cases as well as a broad range of others.

Optimization algorithms. Various constrained optimization algorithms have been employed to solve resource allocation problems, depending on the variable types, objectives, and constraints. For linear programming problems like traffic engineering, the Gurobi solver [22] utilizes two primary methods. The simplex method [46] iteratively progresses along the boundaries of the feasible region toward the optimal solution, while the barrier method [31] iteratively approaches the optimal solution from within the feasible region. In mixed-integer linear programming like load balancing, the branch-and-bound method [7] divides the problem into smaller subproblems, or branches, and then eliminates certain branches based on bounds on the optimal solution at each iteration. For more complex optimization problems, such as cone programming when maximizing proportional fairness in cluster scheduling, the ECOS algorithm [16] adopts a similar iterative process as the barrier methods, with search directions found by a symmetric indefinite KKT (Karush-Kuhn-Tuck) system. All these constrained optimization algorithms operate sequentially and iteratively, and thus introduce bottlenecks in solving time. In contrast, DeDe decouples and decomposes the problem into many parallel subproblems, and thus allows faster problem solving.

Decomposition for scalability. As resource allocation problems grow, efforts have been made to parallelize the iterative process in optimization algorithms for better scalability. NCFlow [1] accelerates traffic engineering by partitioning the network into clusters and concurrently solving the flow allocation problem in each cluster. POP [44] targets "granular" allocation problems where resources are interchangeable. It randomly splits resources and demands into subsets and allocates each subset in parallel. Soroush [43] generalizes the classical waterfilling algorithm for max-min fair allocation and presents parallelizable combinatorial algorithms with fairness guarantees. Teal [65] models traffic engineering with neural networks to benefit from the massive parallelism in GPU. Eason et al. [17] employ the Benders decomposition algorithm to parallelize the design of a specialized cross-layer backbone network, offering performance guarantees for a particular allocation problem. While these prior decompositions rely on restrictive assumptions or problem-specific designs, DeDe offers a general and theoretically grounded approach solution for diverse resource allocation problems.

9 Conclusion

In this paper, we scale real-world resource allocation problems from a different lens, observing that the vast majority of them exhibit an inherent separable structure. This key insight allows us to systematically decouple resource and demand constraints and decompose the original optimization problem into independent and parallel subproblems for allocating individual resources and demands. We implement this approach, DeDe, as a Python package, offering a push-button solution that accelerates large-scale resource allocation. Through extensive evaluation across diverse resource allocation tasks, we demonstrate that DeDe significantly improves scalability while closely approximating the quality of exact solutions.

Acknowledgments

We thank our anonymous reviewers and shepherd for their insightful comments. We also thank Yang Zhou, Justin Chiu, Alexander Rush, and Victor Bahl for their valuable feedback and support. We extend special thanks to Suvan Chatakondu for enhancing our code after the paper's acceptance. This work is supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work is also supported by National Science Foundation grant CCF-2326605 (Expedition in Computing).

References

[1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting Wide-area Network Topologies to Solve Flow Problems Quickly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 175–200, 2021.

[2] Akshay Agrawal, Stephen Boyd, Deepak Narayanan, Fiodar Kazhamiaka, and Matei Zaharia. Allocation of Fungible Resources via a Fast, Scalable Price Discovery Method. *Mathematical Programming Computation*, 14(3):593–622, 2022.

[3] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[4] Achim Bachem, Martin Grötschel, and Bernhard Korte. Mathematical Programming the State of the Art: Bonn 1982. 2012.

[5] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. Parallelism-Optimizing Data Placement for Faster Data-Parallel Computations. *Proceedings of the VLDB Endowment*, 16(4):760–771, 2022.

[6] Dimitri P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[7] Stephen Boyd and Jacob Mattingley. Branch and bound methods. *Notes for EE364b, Stanford University*, 2006:07, 2007.

[8] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends in Machine learning*, 3(1):1–122, 2011.

[9] Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[10] Ulrik Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social networks*, 30(2):136–145, 2008.

[11] Caihua Chen, Bingsheng He, Yinyu Ye, and Xiaoming Yuan. The Direct Extension of ADMM for Multi-block Convex Minimization Problems is Not Necessarily Convergent. *Mathematical Programming*, 155(1):57–79, 2016.

[12] Jingrong Chen, Hong Zhang, Wei Zhang, Liang Luo, Jeffrey Chase, Ion Stoica, and Danyang Zhuo. NetHint: White-Box Networking for Multi-Tenant Data Centers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1327–1343, 2022.

[13] Michael B. Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. *Journal of the ACM (JACM)*, 68(1):1–39, 2021.

[14] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware Database Monitoring and Consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 313–324, 2011.

[15] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[16] Alexander Domahidi, Eric Chu, and Stephen Boyd. ECOS: An SOCP solver for embedded systems. In *2013 European control conference (ECC)*, pages 3071–3076. IEEE, 2013.

[17] John P. Eason, Xueqi He, Richard Cziva, Max Noormohammadpour, Srivatsan Balasubramanian, Satyajeet Singh Ahuja, and Biao Lu. Hose-based Cross-layer Backbone Network Design with Benders Decomposition. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 333–345, 2023.

[18] Epoch AI. Data on Notable AI Models, 6 2024. Accessed: 2025-05-12.

[19] Epoch AI. Machine Learning Hardware, 6 2024. Accessed: 2025-05-12.

[20] Daniel Gabay and Bertrand Mercier. A Dual Algorithm for the Solution of Nonlinear Variational Problems via Finite Element Approximation. *Computers & mathematics with applications*, 2(1):17–40, 1976.

[21] Roland Glowinski and Americo Marroco. Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de Dirichlet non linéaires. *Revue française d'automatique, informatique, recherche opérationnelle. Analyse numérique*, 9(R2):41–76, 1975.

[22] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[23] Magnus R. Hestenes. Multiplier and Gradient Methods. *Journal of optimization theory and applications*, 4(5):303–320, 1969.

[24] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[25] IBM ILOG CPLEX. User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[26] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. AI Benchmark: All about Deep Learning on Smartphones in 2019. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3617–3635. IEEE, 2019.

[27] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 1375–1389, 2023.

[28] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 642–657, 2023.

[29] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[30] Chuan Jiang, Sanjay Rao, and Mohit Tawarmalani. PCF: Provably Resilient Flexible Routing. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 139–153, 2020.

[31] Narendra Karmarkar. A New Polynomial-time Algorithm for Linear Programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[32] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BlastShield. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 325–338, 2022.

[33] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C. Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, et al. OneWAN is better than two: Unifying a split WAN architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 515–529, 2023.

[34] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious Traffic Engineering: The Road not Taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, 2018.

[35] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *15th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–35, 2021.

[36] Lambda. Deep Learning GPU Benchmarks, 2024.

[37] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: Compute Allocation in Hybrid Clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[38] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 553–569, 2021.

[39] Sleiman Mhanna, Gregor Verbič, and Archie C. Chapman. Adaptive ADMM for Distributed AC Optimal Power Flow. *IEEE Transactions on Power Systems*, 34(3):2025–2035, 2018.

[40] Congcong Miao, Zhizhen Zhong, Ying Zhang, Kunling He, Fangchao Li, Minggang Chen, Yiren Zhao, Xiang Li, Zekun He, Xianneng Zou, et al. FlexWAN: Software Hardware Co-design for Cost-Effective and Resilient Optical Backbones. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 319–332, 2023.

[41] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, et al. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[42] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. Minding the Gap between Fast Heuristics and their Optimal Counterparts. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 138–144, 2022.

[43] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. Solving Max-Min Fair Resource Allocations Quickly on Large Graphs. In *21th USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[44] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[45] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 481–498, 2020.

[46] John A. Nelder and Roger Mead. A Simplex Method for Function Minimization. *The computer journal*, 7(4):308–313, 1965.

[47] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 505–520, 2021.

[48] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. DOTE: Rethinking (Predictive) WAN Traffic Engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1557–1581, 2023.

[49] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[50] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 593–612, 2023.

[51] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective Cloud Edge Traffic Engineering with CASCARA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 201–216, 2021.

[52] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective Capacity Provisioning in Wide Area Networks with Shoofly. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 534–546, 2021.

[53] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[54] Reza Takapoui, Nicholas Moehle, Stephen Boyd, and Alberto Bemporad. A Simple Effective Heuristic for Embedded Mixed-integer Quadratic Programming. *International journal of control*, 93(1):2–12, 2020.

[55] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to Route. In *Proceedings of the 16th ACM workshop on hot topics in networks*, pages 185–191, 2017.

[56] Weitao Wang, Dingming Wu, Sushovan Das, Afsaneh Rahbar, Ang Chen, and TS Eugene Ng. RDC: Energy-Efficient Data Center Network Congestion Relief with Topological Reconfigurability at the Edge. In *19th USENIX Symposium on Networked Systems Design and Implementation*, pages 1267–1288, 2022.

[57] Yu Wang, Wotao Yin, and Jinshan Zeng. Global Convergence of ADMM in Nonconvex Nonsmooth Optimization. *Journal of Scientific Computing*, 78:29–63, 2019.

[58] Zhaohua Wang, Zhenyu Li, Guangming Liu, Yunfei Chen, Qinghua Wu, and Gang Cheng. Examination of WAN Traffic Characteristics in a Large-scale Data Center Network. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 1–14, 2021.

[59] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *2023 USENIX Annual Technical Conference*, USENIX ATC '23. USENIX Association, 2023.

[60] Lucien Werner, Adam Wierman, and Steven H. Low. Pricing Flexibility of Shiftable Demand in Electricity Markets. In *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*, pages 1–14, 2021.

[61] Baoyuan Wu and Bernard Ghanem. l_p-Box ADMM: A Versatile Framework for Integer Programming. *IEEE transactions on pattern analysis and machine intelligence*, 41(7):1695–1708, 2018.

[62] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be Late: Optimizing Spot Instance Savings Under Deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 185–203, 2024.

[63] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[64] Zheng Xu, Soham De, Mario Figueiredo, Christoph Studer, and Tom Goldstein. An Empirical Study of ADMM for Nonconvex Problems. *arXiv preprint arXiv:1612.03349*, 2016.

[65] Zhiying Xu, Francis Y. Yan, Rachee Singh, Justin T. Chiu, Alexander M. Rush, and Minlan Yu. Teal: Learning-Accelerated Optimization of WAN Traffic Engineering. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 378–393, 2023.

[66] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, et al. SkyPilot: An Intercloud Broker for Sky Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 437–455, 2023.

[67] Qianyu Zhang, Gongming Zhao, Hongli Xu, Zhuolong Yu, Liguang Xie, Yangming Zhao, Chunming Qiao, Ying Xiong, and Liusheng Huang. Zeta: A Scalable and Robust East-West Communication Framework in Large-Scale Clouds. In *19th USENIX Symposium on Networked Systems Design and Implementation*, pages 1231–1248, 2022.

[68] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2723–2740, 2021.

[69] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. ARROW: Restoration-Aware Traffic Engineering. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 560–579, 2021.

[70] Jun Zhou, Yang Bao, Daohong Jian, and Hua Wu. PDAS: A Practical Distributed ADMM System for Large-Scale Linear Programming Problems at Alipay. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

Appendix

A Evaluation Setup of Cluster Scheduling

We detail the experimental setup for the large-scale cluster scheduling evaluation described in §7.1.1.

Computing resources. We collect 456 distinct GPU/CPU resource types from three benchmark sources [19, 26, 36]. These resource types differ along multiple dimensions such as vendor, interconnect, memory configuration, deployment platform, etc. While it is unlikely today for any single cluster to host all 456 types, recent trends in distributed ML workloads [62, 66] suggest the possibility of scheduling ML jobs across global-scale clusters in the future. For each resource type, the number of available instances is randomly drawn from the set $\{8, 16, 24, \ldots, 64\}$, reflecting common modern hardware configurations where nodes are provisioned with GPU/CPUs in multiples of eight. This setup results in a total of 16,520 GPU/CPU instances in our simulated environment.

Jobs. We generate 2,588 types of ML jobs based on the Notable AI Models dataset [18]. For each ML model, we synthesize multiple job types encompassing both training and inference tasks under various numerical precisions. The number of compute instances requested by each job is sampled from $\{1, 2, 4, 8, 16, 32\}$, in line with common practices in data and model parallelism. Job throughput values are either taken directly from ML benchmarks [19, 26, 36] or, when unavailable, estimated based on each job's FLOP requirements and the computational capacity of the respective hardware.

Scheduling Simulator. Our evaluation employs Gavel's ML scheduling simulator [45], with several adjustments. Jobs are time-sliced across available resource types but are not co-located, due to the increased size and resource demands of modern ML workloads. To ensure that jobs do not grow beyond cluster capacity, we set the average job inter-arrival time to 100 seconds in the Poisson process. We begin the simulation from a steady state, maintaining approximately 1,500 to 2,000 active jobs at any given time. Job allocation decisions are made every 6 minutes, consistent with Gavel's default settings, and the simulation runs for 200 scheduling rounds, or 20 hours.





Quantum Virtual Machines

Runzhou Tao
University of Maryland, College Park

Hongzheng Zhu
University of Maryland, College Park

Jason Nieh
Columbia University

Jianan Yao
University of Toronto

Ronghui Gu
Columbia University

Abstract

Cloud computing services offer time on quantum computers, but users are forced to each use the entire quantum computer to run their programs as there is no way to multiplex a quantum computer among multiple programs at the same time. We present HyperQ, a system that introduces virtual machines for quantum computers to provide fault isolation, better resource utilization, and lower latency for quantum cloud computing. A quantum virtual machine is defined in terms of quantum computer hardware, specifically its quantum gates and qubits arranged in a hardware-specific topology. HyperQ enables quantum virtual machines to be simultaneously executed together on a quantum computer by multiplexing them in time and space on the hardware and ensuring that they are isolated from one another. HyperQ works with existing quantum programs and compiler frameworks; programs are simply compiled to run in virtual machines without the programs or compilers needing to know what else might be executed at the same time. We have implemented HyperQ for the IBM quantum computing service, the largest quantum computing fleet in the world. Our experimental results running quantum programs in virtual machines using the IBM service demonstrate that HyperQ can increase utilization and throughput while reducing program latency, by up to an order of magnitude, without sacrificing, and in some cases improving, fidelity in the results of quantum program execution.

1 Introduction

Quantum cloud computing now makes it possible to obtain on-demand access to quantum computers through cloud-based platforms. These first-generation quantum computers from cloud providers in the form of Noisy Intermediate-Scale Quantum (NISQ) [49] devices heralds the initial shift from classical to quantum computing and the potential realization of practical quantum applications [23]. While classical computers consist of circuits made up of gates that process bits, NISQ devices consist of quantum circuits made up of quantum gates that process quantum bits (qubits). Given their high cost and complexity, cloud computing providers typically only have a few quantum computers available. Nevertheless, they are in high demand by users and IBM recently ran its 3 trillionth quantum program [35].

While cloud computing provides robust software infrastructure that leverages virtual machines (VM) to enable efficient, scalable, and flexible multi-tenant utilization of classical computing resources, the software infrastructure for quantum cloud computing is primitive at best. Users of cloud-based quantum computers submit the quantum programs they want to run to as batch processing jobs, and the cloud service runs the quantum programs on the quantum computer one after another and returns the results. There is no virtualization of quantum computing resources, so each program, no matter how big or small, runs by itself on the quantum computer until it completes, at which point another program is run. This is extremely inefficient given that most programs can only utilize a small number of the qubits available on existing NISQ devices, leaving much of the quantum computer unused. Programs can only utilize a small number of qubits because as the number of qubits used increases, the complexity of the underlying quantum circuit being executed also increases, leading to propagation errors due to hardware noise and loss of fidelity in the results. The utilization problem is exacerbated by the limited number of quantum computers available. Users can end up waiting days for a quantum computing service to return the results of their programs given the high demand for quantum computing resources and the inability of the quantum computer to run more than one program at a time. Cloud services often further limit the number of outstanding program submissions from a user at any given time to only a few, adding further delays to obtaining results from running multiple quantum programs.

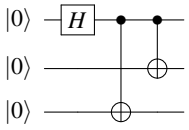
Addressing this inefficiency requires enabling multiple quantum programs to share the underlying hardware. While some research efforts have proposed methods for quantum multiprogramming, they rely on custom quantum compilers to combine multiple programs at compile time [15,34,43–45]. This *static* approach requires knowing precisely which programs will execute together during compilation, preventing independent compilation and execution–a stark contrast to the flexibility enjoyed in classical computing. Many existing techniques struggle with scalability, either limiting the number of co-executing programs or relying on algorithms

with poor complexity for larger machines. Furthermore, these specialized compilers often lack the optimizations found in standard compilation tools like Qiskit.

Inspired by the success of VMs in improving the utilization of classical cloud computing services, we present HyperQ, a run-time management system for cloud-based quantum computing based on a new quantum virtual machine (qVM) abstraction for quantum computers. It virtualizes quantum hardware, i.e., exposes the same interface as a standalone quantum computer, supports multiprogramming *dynamically* and enables fault isolation. It significantly improves resource utilization and reduces latency while mitigating the impact of hardware noise to preserve, and even improve, the fidelity of program execution, all with minimal virtualization overhead.

HyperQ defines a qVM in terms of quantum computing hardware as a quantum circuit abstraction for running quantum programs with a well-defined set of quantum gates and a fixed number of qubits arranged in a specific topology. To enable direct execution of qVMs, the qVM qubit topology must be matched to the qubit topology of the underlying quantum computer hardware. However, qubits can in theory be connected in arbitrary topologies, making it difficult to match the topology of a qVM with that of the quantum computer. Our insight is that we observe that all real quantum computers define a qubit region in which qubits are connected together in a fixed topology, and that region is replicated in a grid-like manner to constitute the overall quantum computer. We leverage this insight to define qVMs in a architecture-specific manner such that the qVMs for a quantum computer are some fixed multiple of its qubit region and thus can be easily mapped to the computer to execute. HyperQ also leverages the natural separation of qubits in the hardware topology to isolate different qVMs and mitigate crosstalk noise when multiple quantum programs are executed simultaneously.

HyperQ virtualizes and manages quantum hardware in a manner that is compatible with existing quantum cloud computing services. Since quantum computer hardware does not yet have quantum memory (QRAM), it lacks context switching capabilities. To support qVMs without being able to save and restore quantum execution state, HyperQ interposes on how quantum programs are submitted to cloud computing services. Users locally compile their quantum programs and submit them in the same manner as they do with existing infrastructure, but the compiler backend transparently targets a qVM instead of a quantum computer. HyperQ encapsulates the compiled executables in their respective qVMs, then schedules qVMs for execution by binpacking them into composite quantum circuits, each of which fits into the qubits of the actual quantum computer. Each composite quantum circuit appears to the cloud service like any other other job, and hence can be submitted to, and executed by, existing quantum computing cloud services. Running a composite quantum circuit enables its qVMs to execute in parallel and in isolation on different qubits of the quantum computer.




```
qubit q[3];
h q[0];
cx q[0], q[2];
cx q[0], q[1];
```

Figure 1: A simple GHZ circuit. ⊕ denotes a CX gate.

To construct the composite quantum circuits, HyperQ optimizes the allocation of quantum hardware resources and partitions resources among qVMs to provide fault isolation. HyperQ uses a simple scheduling algorithm that can guarantee better resource utilization and lower latency for every quantum program compared to the service it would receive without HyperQ on an existing quantum cloud computing service. The algorithm solves a three-dimensional binpacking problem across both space and time while accounting for quantum-specific resource constraints including noise and measurement. It can further tradeoff utilization for improved fidelity of results of quantum program execution.

HyperQ is compatible with existing quantum compilers such as Qiskit [22], enabling quantum programs to leverage its many compiler optimizations. A qVM appears as a backend compilation target similar to actual quantum hardware. Quantum compilers need not know what else will run with the program being compiled, but can simply compile quantum programs to their respective qVMs independently of one another. This is in contrast to existing quantum multiprogramming methods [15, 34, 43–45] that require custom compilers and knowing the exact circuits to be co-executed to combine them at compile time. Since programs are compiled to qVMs instead of the larger underlying quantum computer, they compile faster as compilation time is proportional to the number of qubits of the compilation target, irrespective of whether the program actually uses the number of qubits available.

We have implemented HyperQ and demonstrate that it can be easily used by existing quantum programs and compiler infrastructure, and deployed with existing quantum cloud computing services. Our experimental results with the IBM Quantum cloud computing service and its 127-qubit Eagle quantum computer [50] show that, compared to IBM Quantum, HyperQ can increase resource utilization and reduce latency in obtaining results by up to an order of magnitude for a wide range of quantum programs. We show that qVMs are isolated from one another during execution so that these performance gains can be achieved without compromising fidelity in the program results. Surprisingly, we also show that HyperQ can schedule qVMs in a noise-aware manner that can even improve fidelity in the program results compared to IBM Quantum, while still providing much better utilization and latency.

2 Background

We first provide some background about quantum programs, computers and compilers. For further details, see [21, 42].

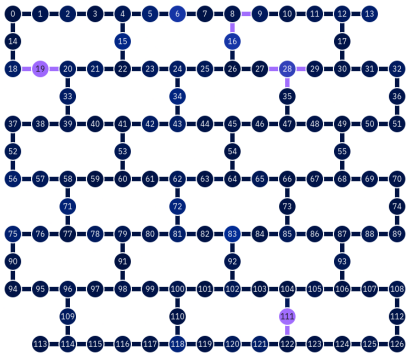


Figure 2: IBM 127-qubit Eagle machine [50].

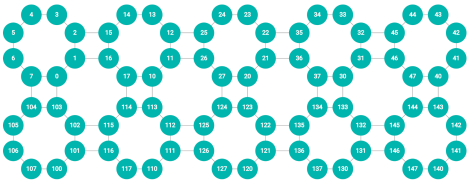
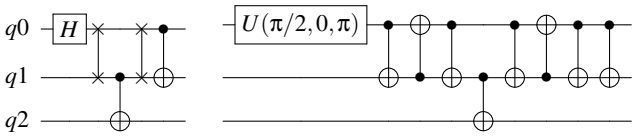


Figure 3: Rigetti 80-qubit Aspen M-3 machine [1].

Qubits, quantum gates and quantum programs In quantum computing, each quantum program is represented as a quantum circuit. The basic unit of a quantum circuit is a qubit, analogous to a register in classical computing. While a bit in classical computing can only be 0 or 1, a qubit can have many more states, specifically any state that is a superposition of 0 and 1. The basic operation on qubits is through a quantum gate, analogous to an instruction in classical computing. A typical quantum gate operates on either one or two qubits. On a quantum machine, two single-qubit gates can run simultaneously on two qubits, as can two two-qubit gates on two disjoint pairs of qubits. Quantum gates manipulate qubits to cause interference, whereby certain qubit states constructively combine and others are eliminated, and entanglement, whereby a qubit's state is correlated with other qubits. Qubits can be measured to get classical bits as result, where the result of a measurement is probabilistic and will collapse the qubit to the state that agrees with the measurement, which will be either 0 or 1. A typical quantum program will perform measurements at the end to produce output of the same number of classical bits as it has qubits. Figure 1 shows a simple GHZ circuit [20] represented by graph and by code.

Quantum computers A quantum computer has a quantum processing unit (QPU) that supports a fixed set of quantum gates, known as its gate set, and arranges its qubits in a particular topology, the latter being described by a coupling map. Similar to classical logic where a set of AND, OR, and NOT gates are sufficient, in quantum computing, there is also a universal gate set. A common universal gate set is the single qubit rotation and an entangled two-qubit gate such as the Controlled-NOT (CX) gate, shown in Figure 1. Different quantum computers may support different gate sets, or instruction sets, and qubit topologies. For example, Figures 2 and 3 shows the coupling maps of the IBM Eagle and AWS Rigetti Aspen M-3, which are quite different. Quantum computers are also equipped with classical registers to store the results of measurements. Note that quantum computers currently lack QRAM, so there is no mechanism to save qubit states outside of the QPU. The only option is to reduce qubits to classical bits by measuring them so the results can stored in classical RAM, with the resulting loss of quantum superposition.



(a) After routing. **(b)** After nativization.

Figure 4: Compilation of GHZ circuit. In the routing stage, if qubit $q0$ is not connected to $q2$ in the coupling map, it is not possible to have a CX gate between them. Instead, two SWAP gates between $q0$ and $q1$ will be inserted and the CX gate will be applied between $q1$ and $q2$. In the nativization stage, with the instruction set $\{\texttt{U},\texttt{CX}\}$, H is translated into $\mathrm{U}(\pi/2,0,\pi)$ and SWAP is expanded into three CX gates.

Quantum compilation Quantum compilation transforms a quantum program into a quantum circuit that satisfies the topology and gate constraints specified by a quantum computer. This includes mapping the qubits used by the program to actual qubits available in the quantum computer. A routing stage [26, 36, 57] inserts SWAP gates into the circuit, to swap qubits, so that two-qubit gates are only applied to qubit pairs that are actually connected in the coupling map. A nativization stage [33] decomposes all quantum gates into those in the instruction set of the target quantum hardware. Figure 4 shows an example for compiling the GHZ circuit. Because compilation is based on the constraints of the target quantum hardware, the resulting compiled quantum circuit is only for the target hardware and is generally incompatible with other hardware with different instruction sets and qubit topologies. For example, a quantum program compiled for the IBM Eagle will not run on the AWS Rigetti.

Noise A significant challenge in near-term quantum computing is the presence of noise. Noise arises from unwanted interactions between the qubits and their surrounding environment, such as thermal fluctuations or electromagnetic interference when performing quantum operations and measurements. Noise introduces errors into quantum operations, leading to a decrease in the fidelity of quantum states and ultimately causing the execution of quantum programs to yield incorrect results. Noise can occur from quantum gates and measurements, as well as crosstalk between qubits. Crosstalk occurs when operations on one qubit unintentionally affect the states of neighboring qubits. It is typically caused by unintended electromagnetic coupling or leakage of control signals be-

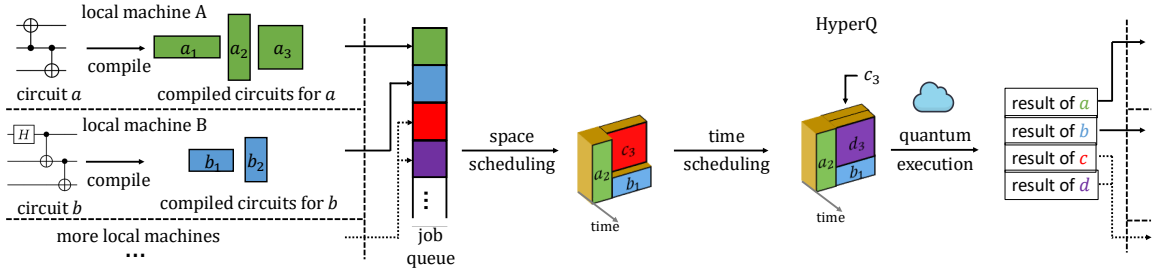


Figure 5: HyperQ workflow.

tween nearby qubits on the quantum chip, and its effects are strongest between qubits that are directly connected with one another. Crosstalk is a major source of noise for NISQ devices and its impact can be an order of magnitude greater than gate errors; mitigating crosstalk is critical to achieve reliable and accurate quantum computation on NISQ devices [37, 45].

3 Design

HyperQ is designed to be compatible with existing quantum programs, computers, and compilers. It accomplishes this based on the observation that quantum computers are structured using a basic repeating region of qubits. For example, the IBM Eagle in Figure 2 is structured around a repeating I-shaped region of qubits, while the Rigetti in Figure 3 is structured around a repeating octagonal-shaped region of qubits. By leveraging this repeating structure, HyperQ can provide qVMs such that programs can be independently compiled to run in qVMs, which are in turn placed into these regions on the machine and executed correctly because the topology and gate set of the qVM matches that of the real machine.

Figure 5 shows the workflow of HyperQ. First, users write and compile their quantum programs on their local machines, where the compiler target is a qVM instead of the actual quantum hardware. Second, the resulting compiled quantum circuits are sent to HyperQ to schedule and run in qVMs on the actual quantum computer. Because current quantum computers do not support context switching, HyperQ schedules and multiplexes qVMs on the hardware by binpacking them in space and time into batch jobs of some estimated execution duration. Each job is created by composing the qVMs in the batch together into one big composite quantum circuit. The batch jobs are then submitted to the cloud quantum computing service to be executed on the quantum machine. After a batch job is completed, HyperQ will receive the measurement result of the combined quantum circuit, which contains results for all qVMs. HyperQ will postprocess the measurement result to identify the part that belongs to each qVM, and return the separated results to the users.

Like VMs on classical computers as originally described by Popek and Goldberg [47], HyperQ ensures that qVMs meet three virtualization properties of interest whenever a quantum program is run in a qVM: efficiency, resource control, and equivalence. The efficiency property states that all innocuous instructions of the program are executed by the hardware directly, where an innocuous instruction is one that does not depend on its location in physical memory or the privilege mode of the CPU running the instruction. This implies that HyperQ ensures that the execution speed of a quantum program in a qVM is at worst a modest decrease versus running directly on the underlying hardware. Furthermore, since quantum computers lack QRAM and QPUs lack privilege modes, all quantum instructions are innocuous. The resource control property states that the program cannot affect the system resources available to it. This means that HyperQ retains control of the hardware and ensures that qVMs are isolated from one another. The equivalence property states that any program executing in a virtualized environment should exhibit the same behavior as if the program had run directly on the underlying hardware, with the possible exception of differences caused by timing or availability of resources. This implies that HyperQ retains compatibility with existing hardware and software so that it can run unmodified quantum programs in qVMs and have them execute correctly.

qVMs have other similarities and differences with VMs on modern classical computers. For example, qVMs are architecture-specific, such that a qVM for an IBM Eagle will not work on a Rigetti quantum computer, just as an x86 VM will not run an Arm VM. A key difference is that given the absence of quantum context switching capabilities, HyperQ does not rely on trap-and-emulate virtualization or hardware virtualization extensions to support it.

3.1 qVM Interface

HyperQ provides a qVM interface that is the same as that provided by a quantum machine backend such as IBM Quantum or Amazon Braket. A qVM provides a virtual coupling map and instruction set, much like a quantum machine provides a coupling map and instruction set, also known as its gate set. It is architecture-specific, just like classical VMs. For example, just like a VM for an x86 machine will not run directly on an Arm machine, a qVM for an IBM Eagle machine will not run on an AWS Rigetti machine. Users can construct their quantum programs using any quantum programming frameworks like Qiskit. A user can then compile the developed program

into an executable that can run in a qVM, which means it conforms to the topology constraint of the given coupling map of the qVM and only uses the gates provided by the gate set. The compiled executable for the qVM will include information regarding how the program's qubits map to the virtual qubits of the qVM, and identification of the set of classical bits the program uses for output.

Basic qVM configuration A basic qVM is defined based on the repeating structure of a specific physical machine. Its coupling map is a shape that is repeated in horizontal and vertical directions in the target physical machine. For example, for the IBM's Eagle machine, we define the basic qVM's coupling map to be an I-shape with seven qubits, which is not only the repeating region for the Eagle machine, but also serves as the entire machine configuration for IBM's smaller 7-qubit Falcon machine, shown in Figure 6. In other words, the I-shape is an architecture-specific region for this family of IBM quantum machines. For the Rigetti Aspen M-3, the coupling map is defined as an octagon shape.

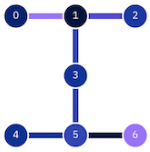


Figure 6: IBM's 7-Qubit Falcon machine [50].

HyperQ splits the physical machine's coupling map into a grid of regions, with each region in the grid have the same shape of the coupling map of a basic qVM, together with some optional connecting qubits between two adjacent regions. For example, the I-shaped region pattern appears in the Eagle machine multiple times. We can split the machine into 9 non-overlapping "Falcons", which form a 3×3 grid, as shown in Figure 7. The instruction set of a qVM is the same as that of the physical machine that will be used to execute the qVM. This is similar to how classical VMs reflect the instruction set of the hardware that they virtualize, except that the qVM instruction set is defined in terms of quantum gates. For the family of IBM quantum machines, the qVM interface is identical to that of the IBM Falcon, and therefore the qVMs can run quantum programs compiled for the IBM Falcon without modification, satisfying the equivalence property. Furthermore, an IBM Eagle machine can run the same qVMs as an IBM Falcon since they have a common architecture that is being virtualized in terms of their instruction set and repeating structure.

Fault isolation HyperQ ensures that qVMs are isolated from one another by also leveraging the structure of the physical machine being virtualized. A key issue that HyperQ must address in this context is that quantum hardware has crosstalk noise, where the state of one qubit can be affected by an instruction applied simultaneously on an adjacent qubit. Crosstalk noise could potentially result in the execution of one qVM affecting the execution of another. HyperQ isolates qVMs from one another by enforcing that qubits from different qVMs are not directly connected to each other. An easy way of doing that is including unused qubits between qVMs to mitigate crosstalk noise. For example, Figure 7 shows that the 9 non-overlapping qVMs have unused qubits between them so that no qVM has a qubit that is directly connected to the qubit of another qVM. As discussed in Section 4.2, our experimental results demonstrate that this simple approach is effective in ensuring fault isolation.

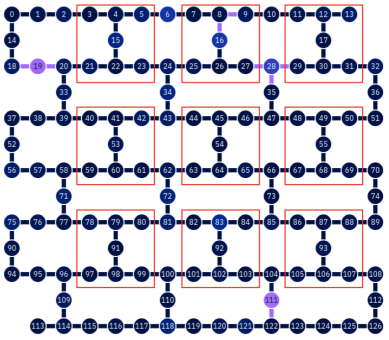


Figure 7: qVM basic unit mapping grid.

Scaled qVM sizes HyperQ provides different sizes of qVMs so that quantum computing resources can be elastically allocated according to computation needs. All the qVM sizes are a multiple of the basic qVM, which we refer to as scaled qVMs. Specifically, HyperQ provides $m \times n$ qVMs, which have the same instruction set as that of the basic qVM, and a coupling map with the shape of an $m \times n$ array of basic qVM shape, together with all the connecting qubits connecting adjacent basic shapes. For example, a 1×2 qVM on the IBM Eagle has 16 qubits and a coupling map of two I-shapes connected horizontally with two connecting qubits.

There are two principles to choose connections for scaled qVMs. First, they need to have the same shape so that we can easily assign the scaled qVMs anywhere on the real machine. For example, we would like to assign scaled qVMs anywhere within the 3×3 grid for the IBM Eagle. Unfortunately, the coupling map is not 90-degree rotational symmetric so we cannot make all connections to be the same. To work around this issue, we define horizontal and vertical connections separately, as shown in Figure 8 for the IBM Eagle. By making sure all horizontal connections and all vertical ones have the same shape respectively, we can still achieve the grid abstraction. We just need to be careful that now a n×m VM and m×n VM are different when n≠m. Second, connections within a scaled qVM should not overlap with connections that can possibly be used by other scaled qVMs. In other words, if we remove a scaled qVM from the grid of regions, adjacent basic regions in the remaining grid should still be able to connect. This does not mean all connections must be completely distinct. We can have overlapping connections within a scaled qVM.

Each scaled qVM of a different size also needs a defined coupling map, built from the basic I-shaped unit, horizontal, and vertical connections. For a connection, we use negative qubit number and add corresponding edges, as shown in Fig-

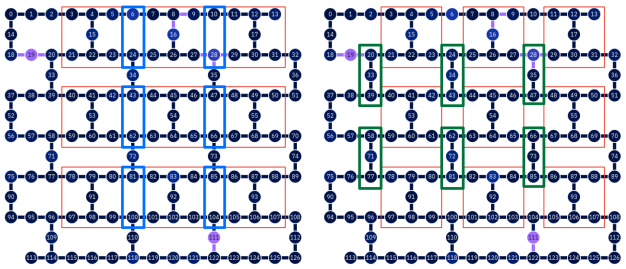


Figure 8: Physical connections for scaled qVMs.

ure 9b. Negative numbered qubits represents qubits inside of connection and positive numbered qubits represent the qubit in the adjacent basic unit. For example, qubit -2 in a horizontal connection should connect to qubit 0 of the basic block on its right, and qubit 2 of the basic block on its left. Each qubit in a scaled qVM needs its own unique number in the coupling map, which we provide by translating the numberings in Figures 6 and 9b based on their relative position. Basic units, horizontal connections and vertical connections are added in row-major order which each component having an assigned qubit number offset. The numbering of a qubit is its component offset plus its number within the component. We ensure that we avoid renumbering the same qubit if horizontal and vertical connections overlap.

HyperQ also scales using fractional multiples of a basic qVM, which we refer to as a fractional qVM. For example, a half qVM for the IBM Eagle would be top or bottom 3-qubit shape in the I-formation, dividing the basic 7-qubit qVM into two fractional 3-qubit qVMs. Fractional qVMs allow multiple small quantum programs to be executed simultaneously in isolation in a region normally taken up by one basic qVM.

For scaled qVMs, HyperQ maintains fault isolation in a similar manner as for basic qVMs. Scaled qVMs that are larger than basic qVMs inherent the same unused qubits between basic qVMs for fault isolation. Fractional qVMs utilizes the unused qubit in the middle of the I-formation between the top and bottom 3-qubit shapes to provide a similar fault isolation mechanism.

Adaptive qVM size selection Unlike classical VMs, a qVM does not have an operating system to multiplex multiple quantum programs because such operating systems are not available, so each qVM runs a single quantum program. Instead of manually selecting the size of a qVM, HyperQ automatically adapts the qVM size to the needs of the quantum program it will run, starting with the compilation process. Quantum programs define the number of qubits used in its quantum circuit, so it is straightforward for HyperQ to rightsize the qVM required. There is no reason to choose a smaller qVM because it will not be able to run the program, and there is no reason to choose a larger qVM because it will occupy additional qubits that it does not need.

For a given qVM size, multiple shapes may be possible.



(a) Horizontal connection **(b)** Vertical connection

Figure 9: VM connection coupling map for scaled qVMs.

For example, a 4 basic unit qVM could be a 1×4 qVM, a 2×2 qVM, or a 4×1 qVM. HyperQ aims to choose the shape that uses the minimal number of basic regions and the best connectivity. By default, HyperQ iterates through all possible (m, n) combinations and chooses the one that can satisfy desired qubit count with the smallest m×n. When multiple configurations have the same m×n, HyperQ chooses the ones with the smallest $|m-n|$. For example, HyperQ would prefer 2×2 over 1×4 since the former has better connectivity. In the default case, HyperQ generates at most 2 versions of qVMs, which can occur when there are two qVMs with the smallest $|m-n|$. For example HyperQ would generate both 1×3 and 3×1 qVMs if 3 basic units are required. In the scheduling step, HyperQ can adaptively select the best qVM shape to use given real-time traffic on the system.

3.2 Compilation

Users compile their quantum programs for qVMs in the same manner as they do for real hardware. There is no difference from the perspective of the user's workflow. Like classical computers, there is no need for the compiler to know what else will be running on the quantum computer as part of compilation. HyperQ works with existing quantum compiler frameworks to automatically select the right size qVM for the program based on the number of required qubits defined in the program. However, a compiler requires a backend to target for compilation. From the perspective of the compiler, each qVM of different dimension is considered a different backend with its own coupling map, just like IBM Falcon and Eagle machines are also considered different backends for existing quantum compilers. To utilize existing compilers, we provide a virtual backend object matched to the respective qVM's coupling map. When compiling a quantum program for a qVM, HyperQ determines the right size qVM based on program information, then selects that virtual backend for the compiler to use to generate the executable quantum circuit. A virtual backend appears just like a backend for real hardware, so existing compilers can work with qVMs without modification. For example, the compiler can perform optimizations on the virtual backend just like it does for a hardware backend. Although this provides broad compatibility with existing compilers, one limitation can be if the virtual backend does not provide the same degree of hardware specificity. In this case, certain highly specialized hardware-specific optimizations that rely on the full knowledge of the physical machine's low-level properties might not be fully exploited during compilation targeting a qVM.

We implemented virtual backends for the Qiskit compiler framework, which is widely used and is the default compiler infrastructure for the IBM Quantum Platform. A Qiskit backend also includes `run` instructions for how to run the compiled executable. In the case of qVMs, the `run` instruction calls HyperQ, which in turn performs scheduling before submitting the qVM to quantum hardware for execution. The virtual backend interface has equivalent behavior as real backends provided by IBM.

3.3 Scheduling

Users submit their qVMs with compiled executables to HyperQ. They are analogous to classical virtual appliances, which are VMs prepopulated with the program they will execute. HyperQ provides hypervisor-like functionality by scheduling when to run the qVM, allocating the necessary quantum hardware resources, and isolating qVMs from one another. To be compatible with existing quantum cloud computing services, HyperQ introduces a scheduling algorithm that is guaranteed to perform at least as good as the current approach of running programs sequentially one after another in FIFO order. The idea for HyperQ's scheduler is to consider qVMs for scheduling in FIFO order, but group qVMs in batches where they can run simultaneously by space and time multiplexing the quantum hardware. Each batch is then sent to the real hardware for real execution.

The scheduling algorithm solves a three-dimensional binpacking problem of how to pack qVMs in space and time to run on a quantum machine. An important constraint with existing cloud quantum computing services is that each program runs to completion and is not preemptible. To fit into this framework, HyperQ essentially packs qVMs together in a batch so the batch appears to the cloud service as one big quantum program that uses some number of qubits and runs for some amount of time.

The scheduling algorithm divides this binpacking problem into two stages. First, HyperQ performs a space scheduling algorithm that partitions the quantum hardware resources into basic regions and assigns those regions to qVMs so they can run at the same time. Second, HyperQ performs a time scheduling algorithm that appends some qVMs into the batch created by space scheduling to even out the amount of execution time required across basic regions. Once a batch is completed and sent to the hardware, the scheduling algorithm will repeat the process again on the remaining qVMs that have been queued but not yet scheduled for execution.

Space scheduling Without HyperQ, a single quantum program running on the physical machine will occupy the whole machine, even if it only uses a small portion of qubits in the machine. HyperQ solves this problem via space multiplexing by dynamically partitioning the machine into multiple regions matched to the sizes of the qVMs that need to be executed. The scheduling algorithm selects a set of qVMs from the job queue and assigns them to regions on the real hardware.

Algorithm 1: Space Scheduling

```
Input: L: a list of executables
Output: a list of compiled circuits with their location
selection ← {}
occupied[r][c] ← false, ∀ 0 ≤ r < maxrow, 0 ≤ c < maxcol
for e in L do
    for v in all versions of e do
        for (r, c) in all positions do
            if regions in [r, r+v.width) × [c, c+v.height) are
              all unoccupied then
                add (v, r, c) into selection
                mark all regions used by v occupied
                if all regions occupied then
                    return selection
                break
return selection
```

Algorithm 1 shows the spatial scheduling algorithm. It employs a greedy approach. The algorithm scans through the job queue of qVMs in arrival order. For each qVM, the algorithm checks whether it is possible to assign that qVM to one (or an array of) unoccupied regions on the hardware. If so, the qVM is assigned to those regions and removed from the job queue. Otherwise, the qVM is left unchanged and will be considered for the next batch. If there are multiple possible assignments, the algorithm will always use the topmost leftmost assignment. If there are multiple shapes of qVMs for a given quantum program, the algorithm will try them in order.

A key feature of HyperQ's approach is that the regions assigned have the same shape as the coupling map of the qVM. This enables mapping a qVM and its compiled executable to run on the real machine without the need for the program to be recompiled. Recall that compilation assigns program qubits to qubits for the target machine, which in the case of a qVM, are virtual qubits of the qVM. Once a qVM is scheduled to a specific region of the real machine, each virtual qubit of the qVM is assigned to a physical qubit of the real machine. HyperQ simply replaces the virtual qubits in the instructions of the executable with the physical qubits that will be actually used for execution. This can be thought of as just changing the qubit labels. This process is easy and fast, and allows a qVM to be placed in any region on the real machine.

Noise-aware scheduling In addition to space scheduling, HyperQ offers a noise-aware variant of its scheduling algorithm, designed to leverage up-to-date machine noise information and improve the fidelity of program execution. For instance, on IBM Quantum's Eagle device, noise calibration data for each pair of qubits is refreshed daily as their noise characteristics vary over time. HyperQ aggregates this data to compute an average noise level for each basic qVM region. It then ranks these regions by noise quality and designates the worst n as "low-quality." By avoiding these low-quality regions during scheduling, HyperQ can improve the fidelity of quantum program results. The parameter n is tunable, allowing

the system to balance throughput against fidelity; our experiments show that excluding the three lowest-quality regions yields a measurable fidelity improvement while sustaining good performance.

Beyond machine-level noise considerations, HyperQ also estimates the noise sensitivity of a qVM based on the program it will run. This could be done in several ways. One approach would be for the qVM to have a configuration option that its user can set to indicate whether its program is noise resilient. For example, a user that is using a qVM could indicate that it is noise resilient if it is running a variational quantum algorithm which can handle high levels of noise versus a Fourier transform which is more brittle to noise. Another approach would be to designate certain qVMs as not noise sensitive using a simple heuristic of their gate count exceeding some threshold because then they are likely to have highly noisy results. During scheduling, these high-noise or noise-resilient qVMs are preferentially placed onto the low-quality regions, thus conserving the high-quality ones for more noise-sensitive workloads. To enable this, we add a single condition to the space scheduling algorithm: if the next job in the queue is a normal qVM, it is assigned to regions of higher quality, whereas a high-noise or noise-resilient qVM is placed in low-quality regions. As with basic space scheduling, the jobs are processed in FIFO order, ensuring fairness while accommodating varying noise characteristics.

Time scheduling Besides space-multiplexing to run multiple quantum programs in parallel, HyperQ also uses time-multiplexing to fill in gaps caused by differences in how long different qVMs need to run, which would otherwise leave machine resources idle. When a qVM with a long running program runs in parallel with a qVM with a shorter program, the time scheduling algorithm appends another qVM to the end of the qVM with the shorter program. After executing the qVM with the shorter program, the qubits it used do not need to stay idle while waiting for the qVM with the longer program to finish, but can be used by another qVM.

Algorithm 2 shows the time scheduling algorithm. It starts from the job batch created by space scheduling. It scans the job queue of qVMs by arrival order. For each qVM, the algorithm will try to append it to a region after an already scheduled qVM, or to an array of regions with qVMs of similar execution time. This appending is valid only when the total execution times of qVMs on that region does not exceed the qVM with the longest execution time scheduled in the previous space scheduling step. If there are multiple possible assignments, the algorithm uses the topmost leftmost assignment. After time scheduling, the overall execution time of the whole batch will remain the same.

A requirement of the time scheduling algorithm is estimating how long each qVM needs to run. While in classical computing, estimating the execution time of arbitrary programs is problematic, this is much easier to do accurately for quantum programs. Near-term quantum programs have no loops and no variable memory access times due to caching behavior. Their execution time is entirely determined by the instructions used, which are the quantum gates. Each quantum gate has a known processing time, so HyperQ uses the circuit critical path length to estimate execution time, where path length is weighted by each gate's processing time. Since quantum compilers such as Qiskit account for gate delay as part of compiler optimization, we can obtain gate processing times from Qiskit's backend properties interface. For example, we found that on the IBM Eagle, a single-qubit gate takes 6e-8 seconds (s) and a two-qubit ECR gate takes about 6e-7 s. HyperQ then uses this information to binpack qVMs in time. Future quantum hardware supporting newer dynamic circuits that can incorporate conditional gate execution may make estimating running time harder; we leave this as future work.

Algorithm 2: Time Scheduling

```
Input: L: a list of executables
S: a list of compiled circuits with their location selected by
space scheduling
Output: a list of selected compiled circuits with their
        location
selection ← S
length[r][c] ← 0, reuse[r][c] ← 0,
 ∀ 0 ≤ r < maxrow, 0 ≤ c < maxcol
for v in S do
    for (r, c) in positions used by v do
        length[r][c] ← v.length
for e in L do
    for (r, c) in all positions do
        for v in all versions of e do
            if putting v at [r, r+v.width) × [c, c+v.height)
              does not increase max length and does not
              exceed reuse limit then
                add (v, r, c) into selection
                update length and reuse of all regions used
                  by v
                break
return selection
```

Appending a qVM after another qVM to time multiplex the same set of physical qubits requires first resetting the state of the qubits so that the appended qVM can be executed without being entangled with the state of the previous qVM. Before the reset, the previous qVM needs to perform a measurement of the qubits to obtain its results. The time scheduling algorithm accounts for the potential delay caused by mid-circuit measurement and reset and will leave gaps unfilled if the costs of measurement and reset in the middle of the execution of some circuits would outweigh the benefit of running another qVM. We use Qiskit's backend properties interface to obtain measurement and reset operation execution times. For the IBM Eagle, the measurement and reset operation times are 1.3e-6s and 1.8e-6 s, respectively.

HyperQ's scheduling provides performance guarantees with respect to the serialized FIFO approach used currently by

cloud quantum computing services. That serialized approach runs programs to completion in arrival order and has each program take up the entire machine when it runs. We refer to the time interval in which a program is executed as a time quantum, and can view the operation of a quantum computer as executing programs in sequentially numbered time quanta. HyperQ guarantees that it will always run a quantum program in a time quanta numbered no higher than the time quanta from using the serialized approach. In other words, any quantum program will be scheduled for execution at least as early as the serialized approach. Furthermore, by enabling multiple quantum programs to run in qVMs simultaneously, HyperQ guarantees that the overall utilization of the quantum machine is always at least as good as the serialized approach.

3.4 Executing with Quantum Hardware

After grouping qVMs into batches, HyperQ submits the batches to the actual quantum hardware for execution. Since current cloud services provide no direct instructional-level support for qVMs, we use existing interfaces to run qVMs by aggregating all qVMs in a batch into a big program that can directly run on real hardware. Then, HyperQ submits the big job to cloud platforms, and gets the measurement result which contains results of all qVMs in the batch. Finally, HyperQ splits the result into results of each program. Note that HyperQ is designed to be a fully dynamic runtime system, so the qVM abstraction can be dynamically supported once online runtime control of quantum hardware becomes available.

qVM aggregation To aggregate the qVMs into a big job, HyperQ combines them all into a single composite quantum circuit. This preprocessing involves qubit translation, gate direction adjustment, and insertion of reset instructions.

For space multiplexing, HyperQ translates all instructions on virtual qubits in the programs compiled to qVMs into instructions on physical qubits. HyperQ maintains a qubit mapping between qVM qubits and physical qubits for all shapes of qVMs and all possible allocation positions of the qVMs onto the hardware. Because qVMs map to the hardware in rectangular grid regions, it is straightforward to maintain this mapping by translating qVM grid coordinates. HyperQ uses this mapping to translate all programs in qVMs by changing their virtual qubits into hardware qubits according to the qubit mapping. For example, if a program is using a virtual qubit 1 which is mapped by scheduling to physical qubit 20, HyperQ will translate the instructions with virtual qubit 1 to instructions with physical qubit 20. Using the mapping, translation is quick and easy to do as a linear scan of the compiled executable. The translated instructions will be executed by the hardware directly and have the same execution speed as the original instructions, because the same instruction has the same execution time on different qubits. HyperQ's use of direct execution ensures that it satisfies the efficiency property.

HyperQ must also perform gate direction adjustment. Qubits are connected by edges, and edges can be directional on real hardware. In a two-qubit gate, the two qubits perform different roles. For instance, in a CX gate, one qubit is the control bit and the other is the operand bit. If the edge between the qubits is directed, one qubit can only be the control bit and the other can only be the operand bit. The IBM Eagle machine uses ECR gates as qubit links and they are directed. Even though it uses a repeating I-shaped region, the gate direction for different I-shaped regions may be different. To account for this, HyperQ uses a qVM coupling map with undirected edges. Once a qVM has been scheduled to an actual region on the hardware, HyperQ translates all instructions with undirected edges to directed edges matching the actual regions. Like qubit translation, this is straightforward to do via a simple linear scan of the compiled executable.

For time multiplexing, HyperQ inserts reset instructions and a barrier at the start of each qVM that follows another qVM using the same qubits. The resets occur in parallel, and the barrier ensures they begin simultaneously, helping to minimize reset latency.

As part of its linear scanning, HyperQ detects any mid-circuit measurement or reset operations and flags any qVMs with programs in which the number of such operations exceeds a defined threshold. This rarely if ever occurs. Such qVMs are removed and placed into separate quantum circuits to ensure that they cannot affect the execution of other qVMs.

Finally, HyperQ concatenates all the preprocessed programs into a big quantum circuit, which contains the execution of all the programs in the batch and is ready to run on the cloud. HyperQ's big quantum circuit is designed to successfully execute on NISQ hardware devices even though other large quantum circuits compiled from quantum programs cannot run successfully due to noise. The reason for this is that HyperQ's combined qVM circuit does not perform quantum operations across the qubits in different qVMs whereas quantum circuits from quantum programs generally do and expect their results to be from the entanglement of all of the qubits being used. As a result, HyperQ's combined circuit does not have the high noise behavior resulting from entanglement across all of the qubits.

HyperQ retains control of the hardware so that any program failures cannot corrupt the hardware or affect later executions of other programs. Before submitting the batch to cloud platforms, HyperQ estimates the running time of the batch by adding up the estimated gate time provided by IBM along the longest gate sequence in the batch. HyperQ submits the big job to the cloud with a timeout limit $1.5\times$ of the estimated running time. If the job fails to finish, it will timeout and control of the hardware will return to HyperQ. The combination of timeouts and unused qubits for fault isolation among qVMs ensures that HyperQ satisfies the resource control property.

Result post-processing After executing the aggregated circuit, HyperQ extracts each qVM's results using a maintained

#	Name_#qubits	Description
1	deutsch_n2	Deutsch algorithm w/ 2 qubits for f(x) = x
2	dnn_n2	3 layer quantum neural network
3	grover_n2	Grover's algorithm
4	iswap_n2	An entangling swapping gate
5	qwalk_n2	Quantum walks on graphs up to 4 nodes
6	bchange_n3	Transform single-particle basis of a linearly connected electronic structure
7	fredkin_n3	Controlled-swap gate
8	linsolver_n3	Solver for a linear equation
9	qaoa_n3	Quantum approximate optimization
10	teleport_n3	Quantum teleportation
11	toffoli_n3	Toffoli gate
12	wstate_n3	W-state preparation and assessment
13	adder_n4	Quantum ripple-carry adder
14	trotter_n4	Trotter steps for molecule LiH at equilibrium geometry
15	bell_n4	Circuit equivalent to Bell inequality test
16	cat_state_n4	Superposition of 2 coherent states w/ opposite phase
17	hs4_n4	Hidden subgroup problem
18	qft_n4	Quantum Fourier transform
19	qrng_n4	Quantum random number generator
20	ansatz_n4	Variational ansatz for Jellium Hamiltonian w/ linear-swap network
21	vqe_n4	Variational quantum eigensolver
22	err_correct_n5	Error correction w/ distance 3 and 5 qubits
23	lpn_n5	Learning parity with noise
24	qec_en_n5	Quantum repetition code encoder
25	qaoa_n6	Quantum approximate optimization
26	simon_n6	Simon's algorithm
27	bb84_n8	Quantum key distribution circuit
28	dnn_n8	16-dimension quantum neural network
29	ising_n10	Ising model simulation via QC

Table 1: Quantum program workloads in the small category.

mapping between qVM classical bits and their positions in the big circuit's output. This linear mapping allows HyperQ to split the cloud's returned results into separate outputs for each qVM.

4 Evaluation

To demonstrate the effectiveness of HyperQ, we have implemented it for the IBM Quantum Platform cloud service and evaluated its performance. From the cloud provider's perspective, we evaluate how much HyperQ can improve both throughput and utilization of quantum computing resources. From the user's perspective, we evaluate how much HyperQ can improve latency by reducing delay in program execution, including accounting for any extra virtualization overhead from the offline processing required to schedule and use qVMs. We also evaluate how HyperQ affects fidelity in program execution, including how much noise HyperQ introduces from multiplexing quantum hardware among qVMs and how it compares to the current single execution approach employed by IBM and other cloud services.

4.1 Experimental Design

We use a subset of QASMBench [25], which covers most types of typical quantum workloads in different sizes. There are three categories of workloads in QASMBench: small, medium and large. The number of qubits in each circuit ranges from 2 to 10 in the small category, and 11 to 27 in the medium category. Within the small category's range, today's NISQ machines can compute correct results with relatively high probability. We do not use the large category, because the circuits either require hundreds of qubits or millions of gates, and therefore are not runnable on near-term quantum computers.

#	Name_#qubits	Description
30	sat_n11	Boolean satisfiability problem
31	seca_n11	Shor's error correction for teleportation
32	gcm_n13	Generator coordinate method
33	multiply_n13	Perform 3×5 in a quantum circuit
34	bv_n14	Bernstein-Vazirani algorithm
35	multiplier_n15	Quantum multiplier
36	qf21_n15	Quantum phase estimation to factor 21
37	dnn_n16	Quantum neural network
38	qec9xz_n17	Quantum error correction 9-qubit code
39	bigadder_n18	Quantum ripple-carry adder
40	qft_n18	Quantum Fourier transform
41	sq_root_n18	Compute square root via amplitude amplification
42	bv_n19	Bernstein-Vazirani algorithm
43	qram_n20	Bucket brigade qRAM prototype circuit
44	cat_state_n22	Superposition of 2 coherent states w/ opposite phase
45	ghz_state_n23	Greenberger-Horne-Zeilinger state for max entanglement
46	knn_n25	Quantum K-nearest neighbor
47	swap_test_n25	Swap test to measure state distance
48	ising_n26	Ising model simulation via QC
49	wstate_n27	W-state preparation and assessment

Table 2: Quantum program workloads in the medium category.

We construct two benchmark sets from QASMBench: small-only and small&med. The small-only benchmark contains circuits from the small category, as listed in Table 1. This benchmark is particularly useful to evaluate fidelity because the expected perfect results of the circuits from the small category can be computed by classical simulators. Therefore, we ensure that all circuits in the benchmark contain actual measurements at the end of execution so that we can approximately reconstruct the resulting statevectors to compare with the expected ones. The small&med benchmark contains circuits from both Table 1 and the medium category, as listed in Table 2. This benchmark better evaluates scheduling performance in the presence of more varied qVM sizes and showcases the elasticity of our HyperQ implementation; it is used for throughput, utilization, and latency measurements.

To use the QASMBench programs to provide an approximation of a real-world workload, we construct an arrival order among the programs by randomly putting each program workload into a queue some fixed number of times. Using the same number of repetitions for each workload facilitates later average fidelity calculations. For the small-only benchmark, there are 29 program workloads and each is repeated 5 times resulting in a job queue of 145 program workloads. For the small&med benchmark, there are 49 program workloads and each is repeated 4 times, resulting in a job queue of 196 program workloads.

We run the benchmarks using four platform configurations:

job arrival	configuration	small-only	small&med
all-at-once	IBM Quantum	456 s	683 s
	HyperQ	54 s	178 s
	improvement factor	8.4x	3.8x
	HyperQ space+time	47 s	139 s
	improvement factor	9.7x	4.9x
	HyperQ noise aware	64 s	176s
	improvement factor	7.1x	3.9x
poisson	IBM Quantum	456 s	683 s
	HyperQ	143 s	230 s
	improvement factor	3.2x	3.0x
	HyperQ space+time	143 s	203 s
	improvement factor	3.2x	3.4x
	HyperQ noise aware	152 s	223 s
	improvement factor	3.0x	3.1x

Table 3: Throughput for HyperQ versus IBM Quantum.

job arrival	configuration	small-only	small&med
all-at-once	IBM Quantum	3.3%	7.8%
	HyperQ	28%	35%
	improvement factor	8.6x	4.4x
	HyperQ space+time	35%	46%
	improvement factor	11x	5.8x
	HyperQ noise aware	23%	35%
	improvement factor	7.2x	4.4x
poisson	IBM Quantum	3.3%	7.8%
	HyperQ	10%	26%
	improvement factor	3.2x	3.3x
	HyperQ space+time	10%	28%
	improvement factor	3.2x	3.6x
	HyperQ noise aware	9.7%	26%
	improvement factor	3.0x	3.4x

Table 4: Utilization for HyperQ versus IBM Quantum.

IBM Quantum The current approach used by the IBM Quantum Platform, and other quantum cloud providers, to run each program individually on the entire machine.

HyperQ Use HyperQ to run programs in qVMs with space scheduling.

HyperQ space+time Use HyperQ to run programs in qVMs with both space and time scheduling.

HyperQ noise aware Use HyperQ to run programs in qVMs with space and noise-aware scheduling.

To evaluate performance with different submission queues, we run all experiments using two different arrival models: 1) all jobs arrive at the start in an infinite job queue (all-at-once); 2) jobs arrive as a Poisson process with rate of 1 job per second (poisson).

All experiments run on the IBM Quantum Platform, where each user is limited to at most three jobs in queue. We used IBM's Brisbane quantum computer, which features the 127-qubit Eagle chip and is available for the public. When a program is executed on the hardware, it is run 4000 times by default to get the resulting state distribution. We used a desktop PC with a 3.2 GHz Intel Core i9-12900K CPU and 32 GB of RAM to compile the programs, schedule and aggregate qVMs into big job quantum circuits, and postprocess results. Compilation was done using Qiskit v1.0 with default settings.

4.2 Results

Throughput We measured the throughput when running the benchmarks. This can be quantified by obtaining the actual run time for each job on the quantum hardware, which is reported by the IBM platform, then obtaining the total run time for each configuration by summing the actual run times. We can then compute the throughput by dividing the number of programs executed by the total run time. Since the number of programs executed for each platform is the same, we can report the total run time as an indirect measure of throughput.

Table 3 shows the throughput results. HyperQ significantly outperforms the baseline IBM Quantum Platform by up to almost an order of magnitude. With jobs arriving all at once, HyperQ has a higher throughput improvement factor for the small-only benchmark because it can run more qVMs simultaneously since the programs require smaller qVMs. Even when running larger programs in the small&med benchmark, HyperQ can provide almost five times better throughput than the baseline IBM Quantum Platform. HyperQ provides better throughput when using both space and time scheduling than using only space scheduling or noise-aware scheduling, but all HyperQ configurations significantly outperform IBM Quantum. With a Poisson job arrival, HyperQ's improvement over IBM Quantum is less significant than with the jobs all arriving at once because there are fewer qVMs in the queue that can be scheduled together. However, HyperQ still provides 3 times or more improvement over IBM Quantum for all configurations. Note that HyperQ and HyperQ space+time are the same for the small-only benchmark with Poisson job arrival because there are not enough additional jobs in the queue to use time scheduling. HyperQ noise aware only has 6% lower throughput than HyperQ space+time with a Poisson job arrival because when there are fewer qVMs to schedule, there are a sufficient number of good regions to run the qVMs.

Utilization We measured the utilization rate when running the benchmarks. Traditionally, utilization is defined as the ratio between the number of active qubit seconds and the total number of qubit seconds. Ideally, the total number of active qubit seconds should be the sum of the active seconds of each qubit, however, the IBM platform provides no reporting of how long each qubit is actually performing operations. We approximate this measure by assuming that for each program, all qubits that are used by the program are active for the whole time the program runs on the quantum hardware. As a result, we can approximate utilization using the running time of each program. We estimate that amount by running each program alone and recording the total time IBM reports. For a HyperQ generated job, the number of active qubit seconds is calculated

job arrival	configuration	small-only						small&med					
		compile	schedule	queue	run	total	improve	compile	schedule	queue	run	total	improve
all-at-once	IBM Quantum	0.04	N/A	226	3.2	229	N/A	0.22	N/A	338	3.5	342	N/A
	HyperQ	0.04	0.14	26	3.6	30	7.6x	0.16	0.40	96	5.1	101	3.4x
	HyperQ space+time	0.04	0.26	22	5.2	28	8.2x	0.16	0.70	76	6.6	83	4.1x
	HyperQ noise aware	0.04	0.14	33	3.6	37	6.2x	0.16	0.49	96	5.0	101	3.4x
poisson	IBM Quantum	0.04	N/A	156	3.2	159	N/A	0.22	N/A	244	3.5	248	N/A
	HyperQ	0.04	0.04	1.2	2.5	3.7	43x	0.16	0.29	7.0	5.0	12	20x
	HyperQ space+time	0.04	0.04	1.2	2.5	3.7	43x	0.16	0.34	4.4	5.0	9.7	26x
	HyperQ noise aware	0.04	0.04	1.4	2.6	4.0	40x	0.16	0.34	9.6	4.7	15	17x

Table 5: Average latency for HyperQ versus IBM Quantum (seconds).

as the sum of the qubit seconds for all qVMs.

Table 4 shows the utilization results. Similar to the throughput results, HyperQ far outperforms the baseline IBM Quantum Platform, by up to an order of magnitude. With jobs arriving all at once, HyperQ has a higher utilization improvement for the small-only benchmark than the small&med benchmark because it is easier to pack more qVMs together when they are mostly of similar size. Nevertheless, HyperQ achieves relatively high utilization for both benchmarks, demonstrating its ability to maintain good utilization independent of what workloads are executed. With a Poisson job arrival, HyperQ's improvement over IBM Quantum is less significant than with the jobs all arriving at once because there are fewer qVMs in the queue that can be scheduled together. However, HyperQ still provides over 3 times improvement over IBM Quantum.

While HyperQ's utilization is many times higher than current practice, its utilization is limited by external fragmentation: the number of qVM qubits that can be used out of the total number of qubits on the machine. At most 85 of the 127 qubits are used for the 127 qubit IBM Eagle when all 9 qVM regions are in use. If the number of jobs is small, not all qVMs are used. For example, for the small&med benchmark, external fragmentation is 59% for all-at-once and 33% for Poisson arrival.

Latency We measured the latency for running the benchmarks with each platform configuration. In the baseline IBM Quantum case, users need to compile the program, submit it to the cloud service, wait in queue, and finally the program can be executed. With HyperQ, there is also the added step of scheduling qVMs and aggregating them into a job that can be submitted to the cloud service. Time spent waiting in queue is the most variable component of latency, and depends on how much work has been submitted to the cloud service across all users. To provide a conservative measure of queue waiting time that ignores jobs from other users, we assume that the queue is initially empty and there are no jobs from other users. The resulting queue waiting time for a job is then only due to the latency of running other jobs in the benchmark before it. With HyperQ, because all programs in a batch job do not have their results returned by the cloud service until the entire batch is complete, we use the run time of the entire batch job as the measure of run time for each program that is run simultaneously in the batch.

Table 5 shows the latency results on average across all programs. HyperQ significantly reduces the average latency of program execution by up to forty times. The key reason for this is because it reduces the queue waiting time because of faster consumption of the program workloads for execution by running them simultaneously. The queue waiting time is the dominant factor in latency, even with the assumption that the queue is initially empty, so that is the most crucial component of latency to reduce. HyperQ adds scheduling overhead from having an extra scheduling step, but it is negligible. HyperQ also increases average run time because short programs running in a batch need to wait for the longer programs in the same batch to finish. However, our results show that the increased scheduling and running time are negligible compared to the queue waiting time.

In practice, the waiting time for running the benchmarks also depends on jobs submitted by other users as well, further increasing queue waiting time. This waiting time for other users depends on the highly varying amount of workload being serviced by the IBM Quantum Platform. There is a real qualitatively different usage experience with HyperQ versus directly using IBM Quantum because of waiting time. With IBM Quantum, running either benchmark takes anywhere from a day or two to an entire week. In contrast, with HyperQ, running the benchmarks typically takes a few hours. Specifically, running the small-only and small&med benchmarks directly with IBM Quantum took 19 and 40 hours, respectively, while using HyperQ typically took 2-3 hours for either benchmark, though sometimes it took as little as 5 minutes.

Fidelity We measured the fidelity of results for running the benchmarks with each platform configuration. It is crucial that HyperQ does not negatively impact fidelity as a result of multiplexing program execution together such that hardware noise from nearby qubit usage for one qVM ends up corrupting the results of another qVM. The result of a quantum circuit can be seen as a probability distribution over states described by binary strings. We use the L1 distance between the distribution given by results of the 4000 runs and the ideal probability distribution on a classically-simulated ideal quantum computer. We calculate the ideal distribution using

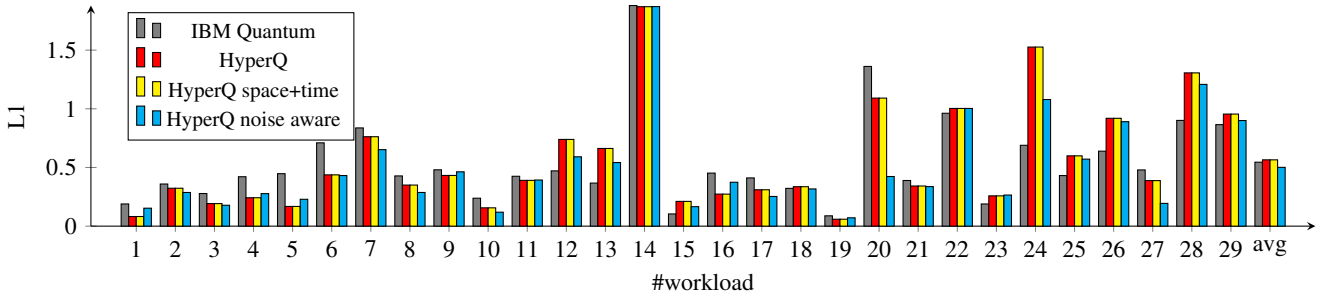


Figure 10: Fidelity for HyperQ versus IBM Quantum per workload with Poisson job arrival.

Qiskit's Aer simulator. The L1 distance is defined as

$$L1 = \sum_{s \in ES} |p_{ideal}(s) - p_{real}(s)| \quad (1)$$

where ES is the set of possible results, $p_{ideal}(s)$ is the probability of result s in the ideal distribution, and $p_{real}(s)$ is the probability of result s in the real result. The value of $L1$ ranges from 0 to 2. $L1 = 2$ means the result is completely corrupted by noise, while $L1 = 1$ means the result has more than half of the perfect result. If a quantum program generates a deterministic result, such as Grover's algorithm or a quantum adder, then $L1 = 1$ is sufficient fidelity to ensure that correct results are generated by the program. We can therefore use this threshold as a rough measure of good fidelity.

Table 6 reports the average fidelity results for the small benchmark. Figure 10 shows the results on a per workload basis for Poisson job arrival. HyperQ provides similar fidelity as the baseline IBM Quantum case of running each program in isolation on the entire machine, adding negligible noise on average for both the space only and noise-aware scheduling. However, HyperQ with time scheduling does incur higher noise for jobs arriving all at once. This is caused by additional mid-circuit measurement and reset operations introduced by time scheduling. These relatively new operations are still noisy on current hardware, though they are expected to improve in future hardware [19, 38] which will make time scheduling more beneficial. Nevertheless, in all cases, HyperQ provides average fidelity that is much better than $L1 = 1$. Overall, HyperQ effectively isolates program execution in qVMs such that space multiplexing them on quantum hardware does not compromise the fidelity of program execution.

Moreover, HyperQ noise-aware scheduling can provide better average fidelity results than IBM Quantum, improving averaging fidelity by roughly 10% over IBM Quantum for Poisson job arrivals. HyperQ noise-aware scheduling does better for Poisson job arrivals than jobs arriving all at once because when there are fewer qVMs to schedule together, there are a sufficient number of good regions to run the qVMs. In contrast, HyperQ space only scheduling does slightly worse for Poisson arrivals than jobs arriving all at once because when not all regions need to be used, inadvertently using a bad region and leaving out good regions to run qVMs worsens average fidelity. Figure 10 also shows various workloads in which HyperQ provides substantially better fidelity results for certain workloads. A closer investigation finds that, for a fixed quantum program, the Qiskit compiler will by default always select the same set of qubits to run the program. If this selection is bad, the result will have bad fidelity. In contrast, HyperQ's space scheduler can assign the program to different regions, averaging out bad choices, while its noise-aware scheduler only puts high noise workloads on qVMs running in noisy regions. Given the degree of isolation that is already provided by qVMs, the region selection can have a more significant impact on fidelity than any potential crosstalk from concurrent execution of qVMs. As a result, HyperQ noise-aware scheduling can surprisingly provide better fidelity results than directly using IBM Quantum.

configuration	all-at-once avg L1	poisson avg L1
IBM Quantum	0.55	0.55
HyperQ	0.55	0.57
HyperQ space+time	0.64	0.57
HyperQ noise aware	0.54	0.50

Table 6: Average fidelity for HyperQ versus IBM Quantum.

HyperQ's ability to provide good fidelity is due in part to ensuring that qVMs are not directly connected to one another. To quantify the impact of using qVMs with no direct connections, we ran a simple experiment with five two-qubit quantum programs, each of which consisted of 100 random quantum X and CX gates. We use two-qubit programs because they are simple but enough to represent the difference between interactions between qubits within a program versus crosstalk between programs. We use 100 gates to have programs that have deep enough circuits to show the impact of propagation errors due to noise. We use X and CX gates because they are deterministic, instead of having their results based on a probability distribution, so that it is easy to simulate such a program on a classical computer to determine its correct result without noise. For each program, we ran it in three configurations on a line-shaped area of the IBM Eagle: 1) single program by itself (qubits 4+5), 2) two copies of the program running on adjacent qubits (qubits 4+5 and 6+7), and 3) two copies of the program running in regions that are separated from one another by one qubit (qubits 4+5 and 7+8). The average success rate of the three configurations in computing the correct

result was 85%, 81%, and 85%, respectively. In other words, having concurrently executing quantum programs on directly connected qubits results in worse fidelity, while using individual unused qubits between such programs maintains the same level of fidelity as when running the program by itself. This result further demonstrates that including individual unused qubits between qVMs is helpful to maintain fault isolation.

5 Related Work

VMs have been used for classical computing since the 1960s, directly following the development of multiprogramming, aiming to ensure noninterference among processes, reduce system complexity for developers, and make software portable across different hardware models [2, 8, 39]. VMs experienced a rebirth in the late 1990s and are widely used on classical computers today [3–6, 9–14, 24, 27–32, 40, 41, 48, 54]. The need to effectively utilize quantum computing hardware and manage multiple circuits running on one machine draws parallels to the desire for VMs after multiprogramming, which inspired our work. HyperQ is the first to introduce qVMs to run on and multiplex quantum computers. While VMs on classical computers benefit from hardware support for virtualization [6], HyperQ enables virtualization of quantum computers and the use of qVMs without additional hardware virtualization features. Although the term "Quantum Virtual Machine" has been used previously for classical simulation of quantum computers [7, 18, 51], that usage does not support any virtualized abstraction on quantum hardware.

The evolving landscape of quantum computing has spurred significant research into quantum software [16, 46, 52, 53], including various custom quantum compilers to support multiprogramming for small-scale quantum machines [15, 34, 43–45]. These custom compilers combine multiple quantum circuits at *compile time* and therefore require knowing at compile time which quantum programs will be executed together. Unlike standard practice on classical computers, they do not allow programs to be compiled independently and later executed in arbitrary combinations. Furthermore, the specialized compilers are usually less optimized compared to using standardized compiler tools like Qiskit. Some of the approaches do not scale to larger quantum machines because they only allow two circuits to run in parallel [15]. None of them improve the fidelity of program execution or consider time multiplexing. HyperQ takes a fundamentally different approach by introducing a powerful virtualization abstraction and manage actual multiprogramming at *run time* by a scheduling algorithm. This preserves independent compilation of programs while enabling scalable multiprogramming, fault isolation, efficient resource utilization, and even improved fidelity of quantum program execution. Furthermore, unlike previous approaches, HyperQ is seamlessly compatible with existing quantum cloud services.

Kernel fusion [17, 55, 56] and program merging [58] have been proposed to improve the utilization of GPUs and programmable network switches by using a custom *compiler* to merge multiple CUDA or P4 programs into one big program. In contrast, HyperQ employs a simple scheduler to combine multiple programs for execution at *run time*. At compile time, each program is compiled individually by users with a standard compiler.

6 Conclusions and Future Work

HyperQ is the first system to introduce quantum virtual machines and use them to enable better resource utilization and lower latency for quantum cloud computing. By leveraging the repeating qubit regions of real quantum machines, HyperQ can define quantum virtual machines in terms of an architecture-specific quantum instruction set and virtual qubit topology, then map them to directly execute on real hardware with no runtime virtualization overhead. Quantum programs can be compiled for specific-sized virtual machines, and HyperQ will binpack them together in batches in both space and time then translate them to aggregated machine-sized quantum circuits so they can be simultaneously executed on a quantum computer. We have implemented HyperQ for the IBM Quantum Platform in a manner that is compatible with the existing cloud service and compilation tools. Our experimental results on real quantum hardware show that HyperQ can reduce program latency as well as increase utilization and throughput by an order of magnitude without sacrificing fidelity in the results of quantum program execution.

HyperQ is designed to virtualize existing quantum computers, but new generations of quantum computers are being developed that are radically different from previous generations as quantum hardware technologies improve. We are exploring how HyperQ can be extended to take advantage of these advances in quantum hardware for diverse qubit technologies like ion traps and neutral atoms, and across different hardware architectures. We also plan to leverage enhanced mid-circuit measurement and control flow for more flexible time-multiplexing. These developments will enable HyperQ to continue to evolve with hardware to provide the system software and virtualization foundations for future quantum cloud computing infrastructure.

Acknowledgments

David Tarrab updated HyperQ to use the latest version of the IBM Quantum Qiskit Runtime API. Gernot Heiser and Henry Yuen provided helpful comments on earlier drafts. This work was supported in part by a VMware Systems Research Award, DARPA contract N66001-21-C-4018, and NSF grants CNS-2052947, CCF-2124080, CCF-2239484, and CNS-2247370. Ronghui Gu is a co-founder of and has an equity interest in CertiK.

References

[1] Rigetti Announces Commercial Availability of their 80 Qubit Aspen-M and a Teaming with NASDAQ to Explore Financial Applications of QC. *ICV TA&K*, February 2022.

[2] Robin J. Adair, Richard U. Bayles, Les W. Comeau, and Robert J. Creasy. A Virtual Machine System for the 360/40. Technical Report G320-2007, IBM Cambridge Scientific Center, May 1966.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Bolton Landing, NY, October 2003.

[4] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, November 1997.

[5] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–51, November 2012.

[6] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.

[7] Rigetti Computing. The Quantum Virtual Machine (QVM). https://pyquil-docs.rigetti.com/en/1.9/qvm.html, June 2018.

[8] Robert J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[9] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, and Jason Nieh. ARM Virtualization: Performance and Architectural Implications. *ACM SIGOPS Operating Systems Review*, 52(1):45–56, July 2018.

[10] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[11] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[12] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.

[13] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM Architecture. *LWN Weekly Edition*, pages 18–22, July 2013.

[14] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.

[15] Poulami Das, Swamit S. Tannu, Prashant J. Nair, and Moinuddin Qureshi. A Case for Multi-Programming Quantum Computers. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO 2019)*, pages 291–303, Columbus, OH, October 2019.

[16] Haowei Deng, Runzhou Tao, Yuxiang Peng, and Xiaodi Wu. A Case for Synthesis of Recursive Quantum Unitary Programs. *Proceedings of the ACM on Programming Languages*, 8(POPL):1759–1788, 2024.

[17] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing CUDA Code by Kernel Fusion: Application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, July 2015.

[18] Google. Quantum Virtual Machine, March 2025. https://quantumai.google/cirq/simulate/quantum_virtual_machine.

[19] Luke Govia, Petar Jurcevic, Christopher Wood, Naoki Kanazawa, Seth Merkel, and David McKay. A Randomized Benchmarking Suite for Mid-Circuit Measurements. *New Journal of Physics*, 25(12):123016:1–17, December 2023.

[20] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. Going Beyond Bell's Theorem. In *Bell's Theorem, Quantum Theory and Conceptions of the Universe*, pages 69–72. Springer, October 1989.

[21] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. Quantum Programming Languages. *Nature Reviews Physics*, 2(12):709–722, November 2020.

[22] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum Computing with Qiskit. https://arxiv.org/abs/2405.08810, May 2024.

[23] Youngseok Kim, Andrew Eddins, Sajant Anand, Ken Xuan Wei, Ewout Van Den Berg, Sami Rosenblatt, Hasan Nayfeh, Yantao Wu, Michael Zaletel, Kristan Temme, and Abhinav Kandala. Evidence for the Utility of Quantum Computing before Fault Tolerance. *Nature*, 618:500–505, June 2023.

[24] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230. Ottawa, ON Canada, June 2007.

[25] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. QASMBench: A Low-Level Quantum Benchmark Suite for NISQ Evaluation and Simulation. *ACM Transactions on Quantum Computing (TQC)*, 4(2):10:1–26, February 2022.

[26] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, pages 1001–1014, Providence, RI, April 2019.

[27] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.

[28] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, pages 1782–1799, San Francisco, CA, May 2021.

[29] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, pages 3953–3970, Vancouver, BC Canada, August 2021.

[30] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 465–484, Carlsbad, CA, July 2022.

[31] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 201–217, Shanghai, China, October 2017.

[32] Jin Tack Lim and Jason Nieh. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, pages 557–574, Lausanne, Switzerland, March 2020.

[33] Sophia Fuhui Lin, Sara Sussman, Casey Duckering, Pranav S. Mundada, Jonathan M Baker, Rohan S. Kumar, Andrew A. Houck, and Frederic T. Chong. Let Each Quantum Bit Choose its Basis Gates. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO 2022)*, pages 1042–1058, Chicago, IL, October 2022.

[34] Lei Liu and Xinglei Dou. Qucloud: A New Qubit Mapping Mechanism for Multi-Programming Quantum Computing in Cloud Environment. In *Proceedings of the 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2021)*, pages 167–178, Seoul, South Korea, February 2021.

[35] The Quantum Mechanic. IBM Quantum Hits Milestone of 3 Trillion Circuits Run. *Quantum Zeitgeist*, December 2023.

[36] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. Noise-adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, pages 1015–1029, Providence, RI, April 2019.

[37] Prakash Murali, David C. Mckay, Margaret Martonosi, and Ali Javadi-Abhari. Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, page 1001–1016, Lausanne, Switzerland, March 2020.

[38] Paul Nation. How to Measure and Reset a Qubit in the Middle of a Circuit Execution. *IBM Quantum Research Blog*, February 2021.

[39] Rob A. Nelson. Mapping Devices and the M44 Data Processing System. Research Report RC-1303, IBM Thomas J. Watson Research Center, October 1964.

[40] Jason Nieh and Ozgur Can Leonard. Examining VMware. *Dr. Dobb's Journal*, 315:70–76, August 2000.

[41] Jason Nieh and Chris Vaill. Experiences Teaching Operating Systems Using Virtual Platforms and Linux. In *Proceedings of the 36th ACM Technical Symposium on Computer Science Education (SIGCSE 2005)*, pages 520–524, St. Louis, MO, February 2005.

[42] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 10th edition, 2010.

[43] Siyuan Niu and Aida Todri-Sanial. How Parallel Circuit Execution can be Useful for NISQ Computing? In *Proceedings of the 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1065–1070, May 2022.

[44] Siyuan Niu and Aida Todri-Sanial. Enabling Multi-Programming Mechanism for Quantum Computing in the NISQ Era. *Quantum*, 7:925:1–34, February 2023.

[45] Yasuhiro Ohkura. Crosstalk-Aware NISQ Multi-Programming. Bachelor's thesis, Faculty of Policy Management, Keio University, 2021.

[46] Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. A Formally Certified End-to-End Implementation of Shor's Factorization Algorithm. *Proceedings of the National Academy of Sciences*, 120(21):e2218775120, 2023.

[47] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[48] Shaya Potter and Jason Nieh. Improving Virtual Appliance Management through Virtual Layered File Systems. In *Proceedings of the 25th Large Installation System Administration Conference (LISA 2011)*, pages 25–38, Boston, MA, December 2011.

[49] John Preskill. Quantum Computing in the NISQ Era and Beyond. *Quantum*, 2:79:1–20, August 2018.

[50] IBM Quantum. Processor Types. https://docs.quantum.ibm.com/run/processor-types.

[51] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A Practical Quantum Instruction Set Architecture. https://arxiv.org/abs/1608.03355, February 2017.

[52] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T Chong, and Ronghui Gu. Gleipnir: Toward Practical Error Analysis for Quantum Programs. In *The 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2021)*, Online, June 2021.

[53] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W Cross, Frederic T Chong, and Ronghui Gu. Giallar: Push-button Verification for the Qiskit Quantum Compiler. In *The 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2022)*, San Diego, CA, June 2022.

[54] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP 2021)*, pages 866–881, Virtual Event, Germany, October 2021.

[55] Mohamed Wahib and Naoya Maruyama. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2014)*, pages 191–202, New Orleans, LA, November 2014.

[56] Guibin Wang, Yisong Lin, and Wei Yi. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing*, pages 344–350, Hangzhou, China, December 2010.

[57] Chi Zhang, Ari B. Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. Time-Optimal Qubit Mapping. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*, pages 360–374, April 2021.

[58] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT 2018)*, pages 98–111, Heraklion, Crete Greece, December 2018.

A Artifact Appendix

Abstract

This artifact provides the source code and scripts to reproduce the experimental results for HyperQ. The artifact contains Python scripts to implement the core HyperQ hypervisor, run

benchmarks against both a baseline IBM Qiskit configuration and various HyperQ scheduling policies, and scripts to retrieve and analyze the performance data, including throughput, utilization, and fidelity. It allows for a comprehensive validation of the claims made in this paper.

Scope

This artifact is designed to validate the performance claims of the HyperQ system. Specifically, it allows for the reproduction of the results in table 3, 4, 6, and figure 10. README.md provides instructions to run the baseline benchmarks using the default IBM scheduler and HyperQ benchmarks with three distinct scheduling policies.

Contents

The artifact consists of Python scripts for core HyperQ functionalities (such as `HypervisorBackend.py`) and for executing the various benchmarks. Scripts dedicated to data retrieval and result analysis are organized into the `getdata/` and `analysis/` directories, respectively. The `benchmark_result/` directory is used for storing benchmark workload files and experimental outcomes, and includes the `result_ideal.txt` file which provides pre-computed ideal state vectors.

Hosting

The artifact is hosted on GitHub https://github.com/1640675651/HyperQ with branch `main` and commit hash `a1c7460`. Future updates will be pushed to the `main` branch, and we encourage you to use the latest version available.

Requirements

The artifact was developed and tested on a standard Linux environment. It requires several software dependencies for its operation: Qiskit, the Qiskit Aer simulator, and the QASMBench benchmark suite. Additionally, a valid IBM Quantum Account with an API token is essential for submitting jobs to IBM's quantum hardware and retrieving results. For complete and detailed instructions on how to set up these dependencies and configure the necessary environment, please consult the `README.md` file provided in the artifact's GitHub repository.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

QOS: Quantum Operating System

Emmanouil Giortamis Francisco Romão Nathaniel Tornow Pramod Bhatotia
Technical University of Munich

Abstract

Quantum computers face challenges due to hardware constraints, noise errors, and heterogeneity, and face fundamental design tradeoffs between key performance metrics such as *quantum fidelity* and system utilization. This substantially complicates managing quantum resources to scale the size and number of quantum algorithms that can be executed reliably in a given time.

We introduce QOS, a modular quantum operating system that holistically addresses the challenges of quantum resource management by systematically exploring key design tradeoffs across the stack. QOS exposes a hardware-agnostic API for transparent quantum job execution, mitigates hardware errors, and systematically multi-programs and schedules the jobs across space and time to achieve high quantum fidelity in a resource-efficient manner. QOS's modular design enables synergistic cross- and intra-layer optimizations, while introducing new concepts such as compatibility-based multi-programming and effective utilization.

We evaluate QOS on real quantum devices hosted by IBM, using 7000 real quantum runs of more than 70.000 benchmark instances. We show that the QOS achieves 2.6–456.5× higher fidelity, increases resource utilization by up to 9.6×, and reduces waiting times by up to 5× while sacrificing only 1–3% fidelity, on average, compared to the baselines.

1 Introduction

Quantum computing promises to solve computationally intractable problems with classical computers [2, 12, 25, 68]. Thanks to remarkable technological advances in materials science and engineering, quantum hardware has become a reality in the form of quantum processing units (QPUs) [19, 27, 69]. Interestingly, QPUs are now readily available in a quantum-as-a-service fashion offered by all major cloud providers [3, 5, 24, 32].

However, QPUs present fundamentally unique hardware-level challenges that cannot be directly mapped to classical accelerator-oriented computing (we empirically detail these hardware challenges in § 3). In particular, QPUs operate in the NISQ-fashion (Noisy Intermediate-Scale Quantum [57]), leading to strict *hardware constraints* and a *non-deterministic* computing platform [50, 61].

More specifically, QPUs are inherently noisy and small in computational capacity, which limits the size of the problems they can solve [57]. Second, the degree of noise differs across QPUs and time, even of identical architecture and model, making it difficult to decide which QPUs should execute a quantum program with good performance [61]. In addition, we can not trivially multi-program multiple quantum jobs on the same QPU to increase utilization without them interfering with each other in undesirable and unpredictable ways [45], severely degrading performance [40]. Last, scheduling quantum jobs on heterogeneous noisy QPUs exhibits a fundamental tension between quantum performance metrics, such as *fidelity* [21], and classical performance metrics, such as utilization and load balance (i.e., resource efficiency).

Unfortunately, these problems are amplified by the current state of software, where QPUs are managed through rudimentary interfaces, despite the ever-increasing demand for these scarce resources [1, 50, 61]. Researchers have proposed specialized approaches to address some of the aforementioned OS and QPU challenges individually, for instance, fidelity without runtime support [4], basic multi-programming via FIFO or random job selection [16], or heuristics-based scheduling [62]. Unfortunately, these systems are designed in isolation, are tightly coupled to specific policies, and lack the architectural flexibility and interoperability needed for cross-stack coordination.

At the same time, in industry platforms like IBM Cloud [32] and AWS Braket [3], multi-programming is not supported, and users must improve fidelity and select QPUs *manually*. However, the noisy, scarce, and heterogeneous nature of QPUs naturally motivates users to select the best-performing QPUs, further amplifying the already high QPU load imbalance and queue waiting times [50, 61].

In total, there is still no operating system that addresses quantum job and resource management in a unified and systematic way, i.e., an architecture that supports composable, cross-layer mechanisms designed specifically for the constraints of quantum computing. Such a system must integrate error mitigation techniques to improve fidelity, fidelity-aware performance estimation to handle the spatiotemporal variability of QPUs, multi-programming to increase utilization through compatibility-aware co-location of quantum jobs, and a scheduler that balances QPU load and reduces queue times—all under a single, unified stack.

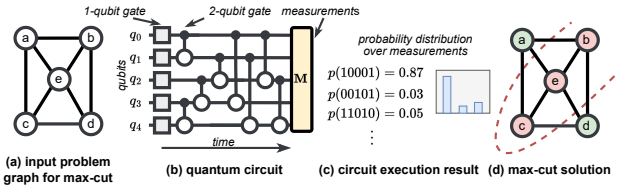


Figure 1: Foundational example (§ 2.1). **(a)** *Input graph to max-cut.* **(b)** *A quantum circuit encoding the max-cut formulation for the graph.* **(c)** *The execution result is a probability distribution of bitstrings.* **(d)** *The result is interpreted as a max-cut between vertices* $\{a,d\}$ *and* $\{b,c,e\}$.

To fill this gap, we propose QOS, a quantum operating system for holistically tackling quantum computing challenges through a modular, cross-layer architecture. At the core of QOS lies the *Qernel* abstraction—the common denominator that enables the seamless composition of diverse system mechanisms and facilitates cross-layer optimizations.

QOS abstracts away the underlying complexity of quantum resource management and systematically explores the associated tradeoffs of quantum computing. To achieve this, QOS exposes hardware-agnostic APIs and comprises four main components: (1) the *error mitigator*, a component that composes complementary techniques to increase fidelity by mitigating hardware noise, (2) the *estimator*, which predicts fidelity performance across heterogeneous QPUs to guide scheduling decisions, (3) the *multi-programmer* that bundles compatible jobs on the same QPU to improve QPU utilization while minimizing fidelity loss, and (4), the *scheduler*, which performs fidelity-aware, multi-objective job scheduling to reduce queueing latency and balance QPU load.

We implement QOS in Python by building on the Qiskit framework [33]. We evaluate QOS on IBM's 27-qubit QPUs [32], using a dataset of more than 7000 quantum runs and 70.000 state-of-the-art quantum benchmark instances used in popular quantum algorithms [38, 60, 76]. Our evaluation shows that the error mitigator improves execution fidelity by 2.6–456.5× on average, depending on the problem size (§ 9.2), the estimator correctly identifies high-fidelity QPUs (§ 9.3), the multi-programmer fidelity by 1.15–9.6× for a target utilization (§ 9.4), and the scheduler reduces the waiting times by 5× while sacrificing at most 3% of fidelity (§ 9.6).

Contributions. Our main contributions include:

1. QOS is the first quantum operating system to holistically address the challenges of quantum computing. Its modular architecture and the introduction of the Qernel abstraction enable a clean separation of mechanism and policy, enabling flexible integration of evolving techniques without modifying system interfaces.
2. QOS enables both cross-layer and intra-layer optimizations through its end-to-end system design. QOS's components synergistically optimize fidelity and resource efficiency while supporting composable techniques within individual layers.
3. QOS systematically explores and manages key quantum computing tradeoffs, including: (i) fidelity vs. runtime overheads in error mitigation and performance estimation; and (ii) fidelity vs. QPU utilization, as well as job waiting times, in multi-programming and scheduling.
4. QOS introduces technical novelties across the quantum software stack: (i) the *Qernel* abstraction as a unifying execution unit, (ii) the composable error mitigation pipeline that is the first to combine circuit cutting, qubit reuse, and qubit freezing—techniques that amplify each other when composed non-trivially, (iii) a multi-programming model introducing the concepts of *compatibility scoring* and *effective utilization*, enabling higher-fidelity circuit co-location, and (iv) the first multi-objective, fidelity-aware scheduler that jointly considers fidelity and waiting times.

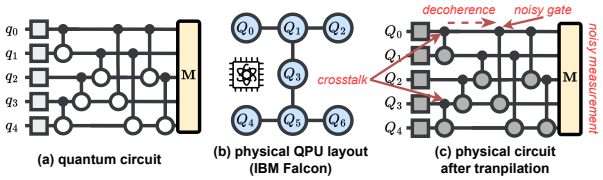


Figure 2: Technical Foundations (§ 2.2). **(a)** *The quantum circuit of Figure 1.* **(b)** *The physical layout of an IBM Falcon QPU.* **(c)** *The transpiled circuit with the QPU's noise sources.*

Artifact availability. QOS is publicly available at https://github.com/TUM-DSE/QOS.

2 Background

2.1 Quantum Computing 101

Quantum computers solve *specific* computationally hard problems exponentially faster than classical computers by leveraging the quantum mechanics principles of *superposition* and *entanglement*. Specifically, the basic units of quantum computation are *qubits*, which during quantum computation are both 0 and 1 at the same time (recall Schrodinger's cat experiment [65]), and when entangled, they can interact with each other even over large distances [41].

To solve NP-hard problems (e.g., maximum-cut) with quantum computers, we use algorithms such as the Quantum Approximate Optimization Algorithm (QAOA) [20]. This algorithm is considered practical for today's quantum hardware capabilities and influential towards achieving quantum advantage [63].

Foundational example. Figure 1 shows how QAOA solves an example max-cut problem for an input graph **(a)**. First, the graph is encoded as a *quantum circuit*. Quantum circuits comprise qubits and quantum gates, akin to logical gates in classical circuits (e.g. NOT, XOR), which are applied over time (from left to right). In the max-cut case, each graph vertex corresponds to a qubit and each edge to a 2-qubit gate in the circuit **(b)**. At the end of the circuit, we measure each qubit to read its value, which gives bitstrings as output. Notably,

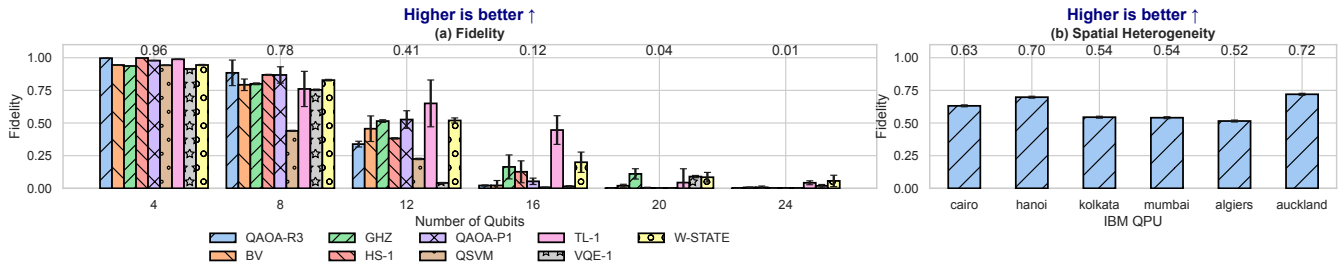


Figure 3: **(a)** Challenge #1, Fidelity (§ 3.1). *Impact of the number of qubits (circuit size) on fidelity. There is an average 98.9% reduction in fidelity from 4 to 24 qubits.* **(b)** Challenge #2, Spatial heterogeneity (§ 3.2). *Fidelity of a 12-qubit GHZ circuit on different IBM QPUs. There is a 38% fidelity difference from best to worst QPU.*

measurements collapse the superposition state to a definite binary value.

Since quantum mechanics is inherently probabilistic, the bitstring we get is truly random, but due to the algorithm structure, it follows a *probability distribution*, which ultimately renders quantum computers useful. Thus, we execute the circuit in many trials (*"shots"*), with each trial providing a specific bitstring from the qubit measurements. The solution of the quantum calculation is, therefore, a probability distribution over all possible bitstrings of the measured qubits, where high probability maps to the solution, while low (~ 0) does not represent a solution **(c)**. In our example, the bitstring `10001` represents the max-cut solution that contains the partitions {a, d} and {b, c, e} **(d)**.

2.2 Technical Foundations

Execution Model. The technology and engineering required to build QPUs renders them an expensive resource, i.e., there are less than 100 QPUs globally offered in the cloud in a quantum-as-a-service fashion [3, 5, 24, 32]. To run quantum programs, users typically write circuit-level code (Figure 2 (a)), which then *transpile* on the QPU to make it executable, send it to the cloud for execution, and finally get the results back. Specifically, the transpilation process performs three key steps: (1) converting the gates of the circuit to the native gate set of the QPU, (2) mapping the logical qubits of the circuit to the physical qubits of the QPU, (3) routing the qubits to the physical qubits with restrictive connectivity by inserting additional costly gates. Figure 2 (b) shows the physical layout of an IBM Falcon QPU. Vertices are the physical qubits, and the edges capture their connectivity, i.e., between which qubits we can apply 2-qubit gates. Figure 2 (c) shows the physical circuit after transpilation with the QPU's noise characteristics, which we detail next.

QPU characteristics. Today's QPUs are described as noisy intermediate-scale quantum (NISQ) devices [57] since they exhibit low qubit numbers (e.g., up to a few 100s [32]) and are susceptible to hardware and environmental noise. Specifically, when measuring a qubit, there is a chance to read the opposite value, and when applying gates, there is a chance the gate performs a wrong operation [23]. On top of that, when qubits are left idle (no gates applied) for more than a few hundred microseconds, the superposition *decoheres* to the $|0\rangle$ state [35], similar to resetting a register to 0. Lastly, qubits destructively interfere with each other via *crosstalk* effects [45]. Figure 2 (c) shows qubits Q_0 and Q_3 that influence each other via crosstalk, noisy gates, qubit Q_0 that is left idle for long enough to decohere, and noisy measurements.

QPU heterogeneity. Additionally, QPUs are vastly *heterogeneous* across space and time, unlike classical accelerators. Across space, QPUs vary in terms of technology, e.g., superconducting qubits [19, 69] or trapped ions [30], architectures of the same technology, e.g., Falcon or Osprey superconducting QPUs [32], and noise properties (formally called `noisemodel`) even for the same architecture [23], e.g., two identical QPUs exhibit different noise errors, etc. Across time, the QPUs are *calibrated* regularly to maintain their performance [31, 77, 82], a process that generates *calibration data*. These data quantify the noise errors and change unpredictably after each calibration cycle.

Performance metric. To measure the quality of a circuit execution on NISQ QPUs, we use the **fidelity** metric [21], which measures the similarity between the noisy probability distribution and the ideal probability distribution that noiseless, ideal QPUs can obtain. Given two probability distributions over measurement outcomes, the ideal P_{ideal} and the noisy P_{noisy} distributions, fidelity is computed as: $F(P_{ideal}, P_{noisy}) = (\sum_i \sqrt{P_{ideal}(i) \cdot P_{noisy}(i)})^2$. Fidelity is a number in the $[0, 1]$ range, where a higher fidelity means a better quality result.

Circuit properties. Execution fidelity is impacted by the aforementioned QPU noise model and the *circuit properties* that describe the circuit's complexity and, thus, susceptibility to errors. Circuit *width* refers to the number of logical qubits involved in the circuit and typically also the physical (H/W) qubits required to execute it. Circuit *depth* denotes the largest number of layers of gates applied to the qubits and quantifies the circuit runtime. A larger depth typically indicates a higher chance for decoherence errors. Last, the number of *non-local gates* refers to the gates that act on non-adjacent qubits, which are particularly susceptible to errors in NISQ devices. *Generally, larger circuits (larger width, depth, or number of*

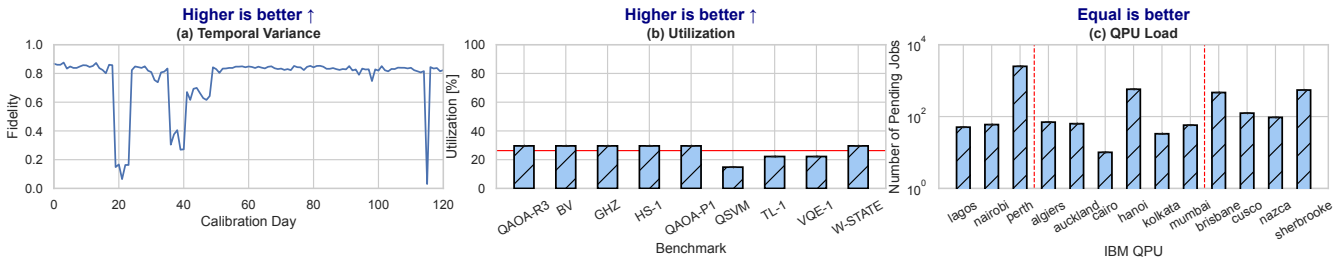


Figure 4: **(a)** Challenge #2, Temporal variance (§ 3.2). *Fidelity of a 6-qubit GHZ circuit on IBM Perth, across 120 calibration days. There are 20 pairs of days with more than 5% difference in fidelity.* **(b)** Challenge #3, Utilization (§ 3.3). *Maximum utilization achieved on a 27-qubit QPU for nine benchmarks while maintaining at least 0.75 fidelity. The average utilization is* 26.3%*, and the max is* 29.6%*.* **(c)** Challenge #4, QPU Load (§ 3.4). *Number of pending jobs on different IBM QPUs. The groups separated by vertical red lines indicate QPUs of the same size. There is up to 57× difference in number of jobs between QPUs of the same size.*

non-local gates) equals lower fidelity.

Circuit feature vectors. While circuit properties are useful, sometimes they fail to characterize the structural and computational properties of quantum circuits. Thus, we can use the Supermarq feature vectors, a set of six metrics proposed by Tomesh et al. [76] with the initial goal of quantifying quantum benchmark coverage. These features quantify the parallelism and speedup potential, the lower bound on execution time, QPU connectivity requirements, entanglement (i.e. complexity), and susceptibility to decoherence and measurement errors.

3 Motivation and Key Ideas

To motivate QOS, we present a set of *unique* challenges that distinguish QPUs from classical accelerators. We categorize our findings into four challenges that must be addressed to improve the practicality of quantum computing: fidelity, utilization, spatial and temporal heterogeneities, and load imbalance. The experimental methodology used is the same for the final system evaluation and is explained in detail in § 9.1.

3.1 Fidelity

Executing quantum programs with high fidelity is challenging since QPUs are characterized by relatively small numbers of qubits and noise, which leads to computation errors (§ 2.2). As the number of qubits and gates in a quantum circuit increases, the noise errors accumulate and the overall fidelity decreases.

Results. Our results are highlighted in Figure 3 (a). The *x* axis shows the circuit size as the number of qubits while the *y* axis shows the fidelity, where higher is better. The experiment is run on the IBM Kolkata 27-qubit QPU. For each increase in qubits, the average fidelity decreases, up to 98.9% from 4 to 24 qubits. Moreover, it is physically impossible to run circuits with a size larger than 27 qubits, since we cannot map them.

Implication. NISQ devices are limited due to size and noise and, therefore, cannot be practically used for large quantum circuits, either logically because the circuit doesn't fit in the device or the execution results would be degraded from noise-induced errors, which translates to low fidelity.

3.2 Spatial and Temporal Heterogeneity

In the classical domain, two identical CPUs perform similarly for all applications at each point in time. In contrast, QPUs exhibit differences in the layout and connectivity of qubits [26] and variations in noise errors even for QPUs of the same model, which leads to spatial performance variance. Moreover, QPUs are calibrated regularly (§ 2.2), and after each calibration, the noise properties change [61]. As a result, the execution fidelity can vary across different calibration cycles, leading to temporal performance variance.

Results. Figure 3 (b) shows a 12-qubit GHZ circuit's fidelity on different IBM QPUs. Fidelity varies across the QPUs, with a maximum difference of 38% from best to worst. Notably, all six QPUs are of the same model (Falcon r5.11).

Figure 4 (a) shows a 6-qubit GHZ circuit's fidelity over 120 calibration days executed on the IBM Perth 7-qubit QPU, where each data point represents a single day's fidelity. The largest single-day difference in fidelity is 96.5%, and there are 20 instances of a single-day fidelity drop of more than 5%. Note that there is no way to predict a QPU's future calibration data to expect such performance differences.

Implications. Due to structural differences across QPUs, quantum circuits perform differently across them. Additionally, there is a high degree of temporal performance variance across calibration cycles, as the fidelity might change significantly from day to day with no discernible pattern.

3.3 Utilization

The fidelity of circuits decreases as their size increases (§ 3.1), and as a result, it becomes more challenging to utilize a QPU effectively. In contrast to the classical domain, where a CPU can be fully utilized, to get high-fidelity results in the quantum domain, we necessarily under-utilize QPUs.

Results. Figure 4 (b) shows the maximum utilization of the IBM Kolkata 27-qubit QPU for nine benchmarks while

targeting at least 0.75 fidelity. No benchmark exceeds 30% utilization, while the average is 26.3%. Higher fidelity targets would yield even lower utilization and vice-versa.

Implications. There is a tradeoff between QPU utilization and fidelity. In general, the lower utilization, the higher fidelity, and vice-versa. In contrast to the classical domain, the tension between these metrics is vastly larger.

3.4 QPU Load Imbalance

The quantum cloud faces QPU load imbalance. The root cause is spatiotemporal heterogeneity (§ 3.2), combined with the manual QPU selection offered by the current quantum cloud model [32]. These naturally motivate users to select the highest fidelity QPU(s) for that calibration cycle [61].

Results. Figure 4 (c) shows the average number of pending jobs for different IBM QPUs across October 2023. The groups of QPUs (separated by the red dashed line) have a size of 7, 27, and 127 qubits, respectively. There is a 49×, 57×, and 5.7× maximum load difference across the groups, respectively.

Implications. Load imbalance leads to long waiting times for the users and thus, low quality of service. However, the performance difference does not always justify the load differences between QPUs. For instance, the 12-qubit GHZ circuit in Figure 3 (b) performs 1.1× better on IBM Hanoi than IBM Cairo, yet the former exhibits 57× higher load.

4 Overview

We propose QOS, an operating system for quantum computing. In this context, we define a quantum operating system as a software layer that transparently manages quantum jobs and resources efficiently with respect to job fidelity and waiting times (user's goals) and QPU utilization and load-balance (cloud operator's goals). Therefore, QOS strives for three design goals: (1) QOS's architecture should be general and modular, cleanly separating mechanism from policy while enabling both cross-layer coordination and within-layer optimizations. (2) QOS should enable the execution of large quantum jobs with high fidelity. (3) QOS should be resource efficient by achieving high QPU utilization and balancing QPU load to minimize queue waiting times.

4.1 The QOS Architecture

Figure 5 shows the overview of our system's design. QOS comprises a layered architecture that consists of an API and four main components: the error mitigator, the estimator, the multi-programmer, and the scheduler, which we detail next.

Qernel abstraction. QOS implements a wide range of mechanisms with different abstraction requirements, from error mitigation to scheduling level. To enable the composability of these mechanisms in a unified architecture,

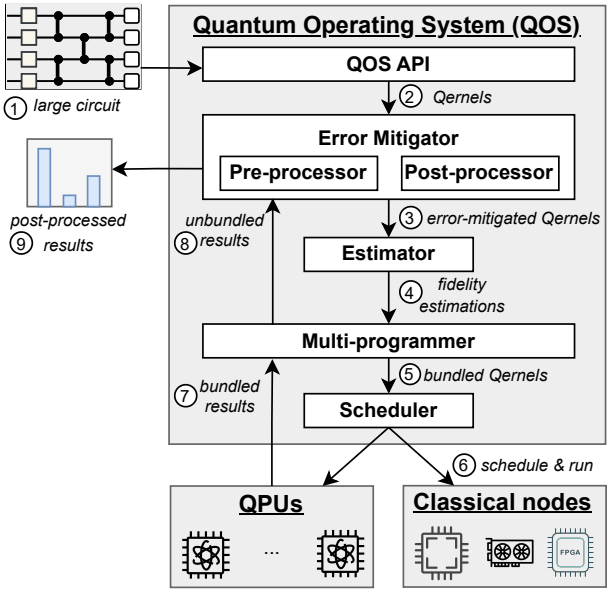


Figure 5: QOS overview (§ 4): *QOS consists of four main components: the error mitigator, estimator, multi-programmer, and scheduler.*

we propose the *Qernel* abstraction that acts as a common denominator for the QOS mechanisms to apply their policies.

QOS API. The QOS API abstracts away the underlying complexity of the noisy and heterogeneous quantum resources by exposing hardware-agnostic functions for configurable quantum job execution. The API transforms the user's input circuits into Qernels for QOS to apply its mechanisms.

Error mitigator. To increase the fidelity of quantum programs, the error mitigator applies pre- and post-processing error mitigation techniques. However, there exists an abundance of such techniques, and they typically incur high runtime overheads that scale with the amount of error mitigation used. Thus, the error mitigator automatically selects and composes a subset of techniques given budget constraints to limit the associated runtime costs.

Estimator. To make informed scheduling decisions in the landscape of QPU spatiotemporal performance variance, we require accurate fidelity estimations that do not depend on expensive simulations or trial runs. The estimator leverages analytical models to predict the execution fidelity on the underlying QPUs in a scalable manner.

Multi-programmer. There is an inherent tradeoff between QPU utilization (proxy of program size) and fidelity. To increase QPU utilization without significantly compromising fidelity, the multi-programmer spatially multiplexes multiple quantum programs on a single QPU with careful consideration for fidelity performance loss.

Scheduler. There is a fundamental tradeoff between job waiting times and fidelity that arises from the underlying spatial performance variance. The scheduler trades minimal fidelity for significant waiting time improvement by optimizing for this conflicting objective.

Table 1: QOS programming API.

QOS API	Description
`run(circ, cnfgs)`	Run circuit with config. options.
`results(jID)`	Retrieve the job results.
`status(jID)`	Retrieve the job status.
`backends()`	Retrieve the available backends.
`backend_props(bID)`	Retrieve the backend properties.

4.2 QOS Programming Model

We introduce the QOS programming model designed to abstract away the underlying complexity of managing heterogeneous and noisy quantum resources. QOS exposes hardware-agnostic APIs and leverages the Qernel unified abstraction that acts as a common denominator across its components to enable the application of its diverse mechanisms.

QOS APIs. Table 1 shows the QOS APIs that abstract away quantum job execution and resource management. To run a quantum circuit, users simply call `run` with the circuit and configuration options, such as the error mitigation budget, which controls the runtime overheads and thus cost (in $). This function returns a unique job id, *jID*. Users can call `result` and `status` with the returned *jID* to retrieve the execution results and status, respectively. Last, users call `backends` and `backend_props` to retrieve the available QPUs and their properties.

The Qernel abstraction. The Qernel implements data structures that store the static and dynamic properties of the quantum job. Specifically, to apply error mitigation techniques optimally, we must leverage the characteristics of the quantum circuit, such as the circuit's size (number of qubits), depth, number and types of 2-qubit gates, the number of measurements, etc. Additionally, we include the six Supermarq features vectors [76] since they are potentially useful for heuristic-based optimizations or regression-based prediction models [59]. This information is described as the static properties of the Qernel. The dynamic properties include the Qernel's execution status (`done`, `failed`, `running`, `scheduled`), the estimator's output, i.e., fidelity estimations (§ 6), and the final post-processed results.

System workflow. Users submit a large circuit using the QOS API ①. Then, QOS transforms the circuit into a Qernel, the common denominator of QOS, and submits it to the error mitigator ②. This component outputs error-mitigated Qernels and submits them to the estimator ③. The estimator predicts the fidelity of running the Qernel(s) on the QPUs to guide scheduling ④. Next, the multi-programmer bundles Qernels with low utilization and sends them to the scheduler ⑥, who schedules and runs the bundled Qernels, optimizing for maximal fidelity and minimal waiting times ⑥. After the execution, the bundled results are unbundled by the multi-programmer into separate results and are sent to the error mitigator for post-processing ⑧. Finally, the post-processor synthesizes the final results and returns them to the user ⑨.

5 Error Mitigator

Quantum computers are characterized by hardware and environmental noise errors, which hinder their practicality and scalability (§ 3). The error mitigator applies pre- and post-processing to the Qernels and execution results, respectively, to mitigate these noise errors.

Challenges. Currently, there is a plethora of individual error mitigation techniques that require their own sub-systems to operate, with no common infrastructure to compose them. However, composability is essential for fidelity improvement since error mitigation techniques can complement each other, if stacked together in the correct order, to further improve performance [9, 11]. Additionally, some techniques spawn an exponential number of sub-circuits and, after execution, require post-processing using classical resources [44, 55]. There are two main challenges in efficiently leveraging error mitigation: (1) Which error mitigation techniques must be used and in which order? (2) How to balance the tradeoff between fidelity improvement and exponential overheads with respect to runtime and resources?

Key ideas. The error mitigator automatically applies error mitigation techniques, abstracting this complexity away from the user. The key idea is that not all gates and qubits are equal; some of them are noise *hotspots*, lowering the fidelity significantly more than others. We constrain the exponential overheads by (1) setting an error mitigation budget b to be spent, and (2) greedily applying error mitigation techniques on the hotspots in the order that maximizes fidelity improvement with the least added overheads.

5.1 Pre-Processor

The pre-processor first analyzes the Qernel to generate important metadata, identifies the error mitigation opportunities, and then prepares the Qernel for the respective techniques.

Qernel analysis. Qernel analysis generates two main pieces of information that guide the application of error mitigation techniques: (1) Qernel metadata and (2) optimization opportunities such as *hotspot* qubits or gates, i.e., qubits and gates that can be removed to reduce noise errors significantly. Specifically, we generate the key Qernel static properties and the six SupermarQ feature vectors as stated in § 4.2.

Error mitigation techniques. We focus on two existing error mitigation techniques: circuit divide-and-conquer and qubit reuse, since these techniques increase fidelity and enable the execution of large quantum jobs on small(er) QPUs. However, QOS supports any technique that achieves the same goals [10, 15, 43, 75].

At a high level, techniques in the former category divide (large) circuits into smaller fragments that can be executed on small QPUs, and the fragmented execution results are post-processed back to a single value. The advantages are two-fold: (1) We can scale the size of circuits that are

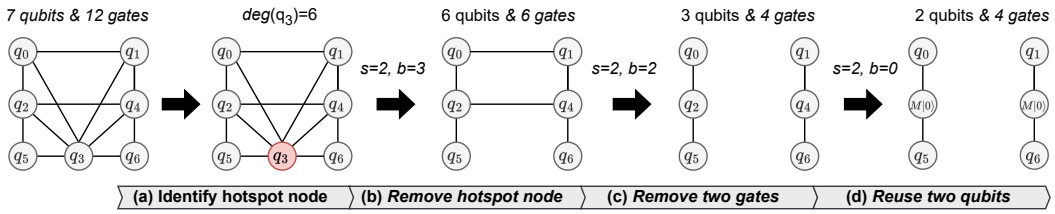


Figure 6: Error mitigator pre-processor workflow (§ 5.1). *The initial Qernel has 7 qubits and 12 gates, and budget $b=3$.* **(a)** *We identify q_3 as a hotspot node with a degree of 6.* **(b)** *We remove this node, which reduces the qubit and gate counts to 6.* **(c)** *We remove two gates, giving two fragments of 3 qubits each.* **(d)** *The budget is depleted, so we use qubit reuse to further reduce the qubits by two.*

executable on the current small QPUs beyond the H/W qubit limit, and (2) we can improve fidelity since smaller and less complex circuits achieve higher fidelity (§ 2.2, § 3.1). Unfortunately, these techniques incur *exponential* quantum and/or classical overheads with respect to the number of divisions; therefore, they must be used conservatively.

More specifically, circuit divide-and-conquer comprises *circuit cutting and knitting* [44, 55] and *qubit freezing* [4]. Circuit cutting can simplify the circuit's structure by virtualizing noisy non-local gates into (less noisy) local gates [44] or cutting a qubit (wire) in the temporal dimension to shorten its runtime [55]. Thus, circuit cutting can improve the circuit's width, depth, and number of non-local gates at once (§ 2.2). During knitting, the original (full) circuit results are reconstructed through classical post-processing of the subcircuit results. We refer interested readers to CutQC [73] and QVM [78] for details on how circuit cutting works in practice. Comparably, qubit freezing identifies qubits with significantly more connections to other qubits (hotspots) and partitions the circuit by replacing the hotspot qubits with binary values, effectively dropping their noisy non-local gates. Since every qubit can be replaced by a binary value, this process generates 2^m smaller circuits for m frozen qubits.

In both cases, we automatically find the cut locations that achieve the smallest sub-circuits possible with the fewest cuts possible. To maximize the synergy between techniques, we first apply qubit freezing to remove multiple non-local gates at once with a relatively small cost, and then we greedily apply circuit cutting. To restrict the exponential overheads that scale exponentially with the number of cuts, we use the budget b to cut up to b times. Typically, we set the budget $b=3$ by default, and the overheads scale as $O(2^b)-O(8^b)$, depending on the divide-and-conquer technique.

On the other hand, qubit reuse (also referred to as recycling) reduces qubit requirements by reusing physical qubits after resetting them [18, 29, 34]. Once a qubit's role in computation is complete, it is measured, reset to a known state (typically $|0\rangle$), and reassigned to another logical qubit later in the circuit. This approach enables more efficient use of limited hardware resources and allows execution of larger circuits than the QPU's qubit limit. This method, however, increases the circuit depth (i.e., execution duration), which can lead to quantum decoherence errors (§ 2.2); therefore, the tradeoff, in this case, is between circuit size and runtime. To restrict the runtime increase, we use qubit reuse as a last resort to render the Qernel executable by at least one QPU in the system (i.e., the optimized Qernel fits at least one QPU's size constraints).

Workflow example. Figure 6 shows the pre-processing workflow for a QAOA circuit (§ 2.1) with 7 qubits and 12 gates. The pre-processor aims to achieve a maximum Qernel size of two qubits with a budget of $b=3$. To achieve this, it takes the following steps:

Step 1: The pre-processor applies analysis to identify a hotspot node; in this case, q_3 is a hotspot with a degree of 6 gates (Figure 6, (a)).

Step 2: Then, the pre-processor removes q_3 and its gates with qubit freezing. The new Qernel size is 6 qubits with 6 gates. Then, it updates the budget to $b=b-m$, where $m=1$ is the number of qubits frozen (Figure 6, (b)).

Step 3: Next, the pre-processor applies circuit cutting on two gates and updates the budget to $b=0$. The new Qernel consists of two fragments, each with a size of 3 qubits and 4 gates (Figure 6, (c)).

Step 4: Since $b=0$, the pre-processor applies qubit reuse to achieve a size of 2 qubits and identifies qubits q_0, q_1 as reusable. Here, $M|0\rangle$ denotes measurement and reset to the $|0\rangle$ state. The final Qernel now has two fragments of 2 qubits and 4 gates (Figure 6, (d)).

The final output is a Qernel with a 42.8% smaller size and 66% less noisy gates. Notably, this result is only attainable through the synergy of techniques, as no individual method alone can achieve comparable improvements.

5.2 Post-Processor

The post-processor reconstructs the final error-mitigated results by classically stitching together intermediate outcomes from sub-Qernel executions. In circuit knitting, each virtualized gate (i.e., cut) is expanded into a linear combination of basis gates with associated coefficients. When multiple such gates are virtualized, the global coefficient vector becomes the tensor product of the individual gate vectors, resulting in an exponentially large space of up to 8^k bitstring-weight pairs for k virtualized gates. The post-processing then requires computing a weighted sum over all 8^k combinations, with each term involving a product of subcircuit results. As such, this

process requires a scalable post-processing infrastructure.

The map phase. To efficiently process the large number of results, we follow a divide-and-conquer approach. Specifically, we split the results into k equal sizes and distribute them to k classical nodes to be processed in parallel. We parallelize across k to increase data locality and reduce communication overheads since all results for each of the k cuts will be in the memory of the same node. Locally, each node performs tensor product ($\otimes$) operations on the probability distributions, which are parallelizable across the node's threads. If available in the node, QOS leverages GPUs or TPUs to accelerate the tensor products. Following this process, the k nodes output k intermediate results, ready to be reduced into a single result.

The reduce phase. QOS selects any of the k nodes to perform the reduce step. The rest of the nodes send the intermediate results to this node, which performs a thread-parallel sum of k results. Equivalent to the map phase, the parallel sum can also be executed on GPUs. This produces the final output to be returned to the user.

6 Estimator

The estimator is responsible for predicting the fidelity performance of a given Qernel on the underlying QPUs without executing the Qernel, which would be extremely costly. This prediction will be the leading decision factor for the scheduler when assigning the Qernel to a QPU.

Challenges. Estimating fidelity a priori faces three key challenges: (1) Fidelity is a non-deterministic metric affected by the hardware's probabilities of errors, which change across QPUs and time unpredictably (§ 3). (2) Simulating a Qernel to obtain fidelity estimates is intractable since the complexity of simulating quantum systems scales exponentially with the number of qubits [22]. (3) Fidelity can be approximated numerically [47]; however, there is an accuracy-cost tradeoff. The more complex the analytical model becomes, the higher the runtime overhead of estimation.

Key idea. The estimator leverages analytical and regression models that do not require real execution or simulation of the Qernel. Specifically, the estimator's models compute a *score* for each Qernel-QPU assignment that captures the potential fidelity of that assignment, and the estimator supports configurable scoring policies with different runtime-cost tradeoffs. These policies consider (1) the Qernels' properties generated from the error mitigator, (2) the QPUs' calibration data (i.e., noise model), which are available to quantum cloud providers since they perform the calibration cycles.

For (1), important properties include the number and types of gates, depth, and the number of measurements (§ 4.2). For (2), recall that QPUs are characterized by calibration data that describe the exact error rates of the QPU for that calibration cycle (§ 2.2), specifically, the individual qubit readout errors, the individual gate errors, and the $T2$ coherence times. In this work, we implement two scoring policies: a numerical approach for fine-grained control over the estimations and a regression model approach for abstracting away the complexity of estimation.

Numerical cost policy. This policy estimates execution fidelity by transpiling the circuit for the target QPU (§ 2.2). Target transpilation enables fine-grained fidelity estimation by producing the mapping between logical and physical qubits and the gate (instruction) schedule. The mapping captures the expected readout and gate errors, while the gate schedule captures the order and exact timing that the gates will be applied on the qubits, which reveals the hardware decoherence and crosstalk errors, as explained in § 2.2. This method has been explored before, and there are multiple variants, but we base our implementation on Mapomatic [47].

To compute the score, we need the following information: (1) The readout/measurement error probabilities that describe the probability of a bit-flip during measurement, (2) the gate error probabilities, (3) the T2 time, which measures for how long a qubit can stay in an excited state (i.e., $|1\rangle$), thus quantifying the decoherence error probability, and (4) the crosstalk error which is measured through crosstalk characterization [45]. With this information, we can multiply the probabilities of the respective sources of errors to get an estimate of the total error/fidelity.

Formally, for each qubit q_i the readout error is $e_{r(i)}$, for each gate g_j the error is $e_{g(j)}$, and the decoherence error is $e_{d(t)} = 1 - e^{-t/T2_i}$, where t is the idle time of the qubit q_i (no gates act on it [14]) and $T2$ is the decoherence time of q_i. The crosstalk error between gates g_k and g_l is $e_{ct(k,l)}$. Putting it all together, the final fidelity score is computed as follows: $fid = 1 - \prod_{i=0}^{N} e_{r(i)} e_{d(i)} \prod_{j=0}^{M} e_{g(j)} \prod_{j=0,k=0}^{M \times M} e_{ct(j,k)}$, where N is the circuit's number of qubits and M is the number of gates. Since all hardware error information is known a priori, and quantum errors accumulate multiplicatively, this policy produces high-accuracy estimations, as we show in § 9.3.

Regression model policy. Since the impact of noise errors on fidelity during quantum computation can be described numerically, we can train a regression model to predict the fidelity of a transpiled Qernel on a possible QPU using the QPU's calibration data and the Qernel's static properties as features. Specifically, we use the aforementioned errors we defined in the numerical cost policy as QPU features and the static properties (§ 4.2) as Qernel features. Even simple regression models such as linear regression achieve high prediction accuracy, up to 99%. Although this policy is simple to use without detailed knowledge of the relationship between errors, in QOS, we use the numerical cost policy by default for estimation to have a clear understanding and full control of the process.

7 Multi-programmer

The size of quantum programs that run with high fidelity is small, leading to QPU underutilization (§ 3.3). To increase QPU utilization, QOS *multi-programs* two or more Qernels, potentially from different users, to run on the same QPU. We refer

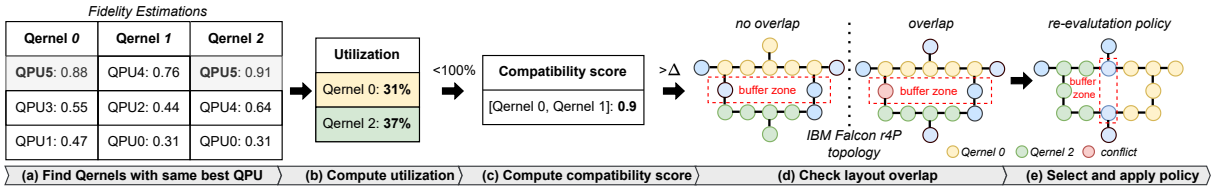


Figure 7: QOS multi-programmer example workflow (§ 7). **(a)** *We use the estimator's output to find Qernels with the same best QPU.* **(b)** *We compute their independent utilization, and* **(c)** *their compatibility score. If compatible,* **(d)** *we check for layout overlap, and* **(e)** *apply the appropriate multi-programming policy.*

to this multi-programming as *bundling* the Qernels together.

Challenges. Bundling Qernels faces three distinct challenges: (1) Trivially bundling Qernels together will deteriorate fidelity because qubits interfere with each other via crosstalk errors (§ 2.2). (2) On top of that, bundled Qernels with unequal runtimes do not optimally increase utilization since *effective* utilization is measured in space and time (number of QPU qubits used and duration that they are non-idle). (3) Qernel compilation involves several NP-hard processes [70], so we need to minimize it.

Key ideas. To tackle these challenges, we introduce new ideas unexplored by prior multi-programming work [16, 40]. First, we define a new utilization metric, namely *effective* utilization, that captures the temporal dimension of utilization, i.e., H/W qubit usage during computation time. Next, we compute Qernel *compatibility functions* that quantify how well-suited any two Qernels are to run together, to minimize fidelity loss and re-compilation times. Last, we create *buffer zones* between two bundled Qernels to minimize crosstalk errors.

7.1 Qernel Compatibility

Qernel compatibility quantifies crosstalk errors and the effective utilization of bundled Qernels by considering the Qernels' static properties (§ 4.2).

Effective utilization. The trivial way to compute utilization in the case of bundled Qernels is by dividing their total number of qubits over the QPU's number of qubits, N_{total}/N_{QPU}. However, this is not accurate when the bundled Qernels have unequal runtimes. In order to accurately quantify QPU usage in the context of multi-programming, we define effective utilization as *spatial* plus *temporal* utilization. To quantify spatial utilization, it suffices to compute the ratio of allocated QPU qubits (of the longest Qernel in terms of depth) over the QPU's number of qubits.

To quantify temporal utilization, we scale the spatial utilization of each allocated Qernel with a weight that is the ratio of the Qernel's duration over the longest Qernel's duration. Recall that, to measure the Qernels' durations, we use the depth property that reflects the longest sequence of gates the Qernel consists of (§ 2.2).

More technically, we define effective utilization as $u_{eff} = \frac{N_{C_{max}}}{N_{QPU}} \cdot 100 + \sum_{k=1}^{n} \frac{D_k}{D_{max}} \cdot \frac{N_{C_k}}{N_{QPU}} \cdot 100$, where $N_{C_{max}}, N_{QPU}$ are the number of qubits of the longest Qernel and the QPU, respectively, k is the number bundled Qernels excluding the longest Qernel, and D is the depth of the Qernel. The first term, $\frac{N_{C_{max}}}{N_{QPU}} * 100$, captures the spatial utilization. The right term (sum) captures the temporal dimension, and $\frac{D_n}{D_{max}}$ is the relative depth of Qernel k compared to the longest Qernel, and $\frac{N_{C_k}}{N_{QPU}}$ is the fraction of QPU qubits used by Qernel k.

For example, a 10-qubit Qernel Q_0 spatially utilizes 50% of a 20-qubit QPU. Now assume that Q_0 runs $3\times$ longer than a 10-qubit Qernel Q_1. During $\frac{2}{3}$ of Q_0's runtime, the qubits allocated to Q_1 will be idle, decreasing the effective utilization to only 66%.

Quantifying crosstalk. To quantify crosstalk without running the bundled Qernels, we use the *entanglement ratio* and *parallelism* Qernel feature vectors of [76], where higher values indicate a higher chance for crosstalk errors (§ 2.2). Intuitively, the entanglement ratio captures the proportion of 2-qubit gates over all gates, and parallelism captures how many gates run in parallel per time unit. Thus, more parallel 2-qubit gates equal a higher chance of crosstalk errors [45].

Compatibility formula. To quantify the compatibility of two candidate Qernels, we consider their effective utilization, as well as their joint entanglement ratio and parallelism scores. This is because we ideally want high effective utilization (not just higher spatial) and fewer crosstalk-induced errors.

Formally, we score a possible Qernel pair as follows: $qc = \alpha\, u_{eff} + \beta\, ER_b + \gamma\, PA_b \mapsto [0,1]$. Higher score is better, $\alpha+\beta+\gamma = 1$ (the weights add up to one), u_{eff} is effective utilization, ER is entanglement ratio, PA is parallelism, and b denotes *bundled*, i.e., E_b is the entanglement ratio of the bundled Qernels. The four variables are tunable to give priorities on different objectives, e.g., prioritize effective utilization or minimize crosstalk. After experimenting and fine-tuning, we found that $\alpha = 0.25, \beta = 0.25, \gamma = 0.5$, and $qc \geq 0.75$ gives balanced results, as we show in § 9.4.

Buffer zone. Crosstalk characterization studies on real machines have shown that the probability of crosstalk errors drops exponentially with the distance between any two qubits [45]. We leverage this to minimize crosstalk errors by creating a *buffer zone* between any two bundled Qernels, i.e., at least one hardware qubit between the two sets of allocated qubits must be free. Given the hardware constraints and limited number of available qubits, we limit the buffer size to up to two qubits of distance.

7.2 Multi-programming Policies

In this work, we implement two multi-programming policies; the first is the *fast path* multi-programming while the second requires re-compilation and re-estimation.

Restrict policy. The restrict policy checks if there is no overlap in the Qernel mappings on the QPU, including the buffer zone. For example, in Figure 7 (d) in the left case, the logical qubits of the Qernels are mapped to disjoint sets of physical qubits on the QPU, and there is a buffer zone between the two sets. In that case, the policy simply bundles the Qernels together, and fidelity loss is minimized through the aforementioned compatibility score.

Re-evaluation policy. This policy is the fallback of the restrict policy. If the Qernel mappings overlap, the two Qernels are transpiled again for the target QPU, and their new fidelity is estimated. If the new fidelity is lower up to a fixed $\varepsilon > 0$ value compared to the original independent fidelities, the bundling is kept. Otherwise, the multi-programmer selects the next most compatible Qernel pair.

Example. Figure 7 shows an example where the multi-programmer receives three Qernels with three estimations each and identifies *Qernel* 0 and *Qernel* 2 as a possible pair since their best QPU is the same (*QPU*5) (a). It computes their independent utilization (31% and 37%, respectively), and the combined utilization is under 100% (b). It computes the compatibility score that surpasses the threshold ($0.9 > 0.75$) (c). In (d), the right case shows an overlap in the buffer zone (the red qubit), so we apply the re-evaluation policy.

Unbundling. To unbundle the results, the multi-programmer keeps a record that maps the initial (solo) Qernel IDs to the new, bundled Qernel ID, as well as the Qernels' sizes. Therefore, when receiving a new result from a Qernel with an ID i, it scans the record to find an entry i, and if found, it splits the probability distribution bitstrings (§ 2.1) into two parts: the left-most and the right-most bits based on the Qernel sizes.

8 Scheduler

Scheduling quantum programs involves fundamental tradeoffs between conflicting objectives; specifically, users ideally want the fidelity of *the best fidelity QPU* and the waiting time of *least busy QPU*.

Challenges. Scheduling quantum tasks faces three key challenges: (1) To maximize fidelity, most programs must be scheduled on the same subset of best-performing QPUs (§ 3.2), forming hotspots and necessarily increasing user waiting times. (2) The scheduler must at least know estimates of the quantum task durations. However, similar to fidelity estimation (§ 6), quantum execution time estimation must be fast, without real execution or simulation. (3) Job scheduling is a well-known NP-hard problem; therefore, every heuristic or greedy solution will present a tradeoff between optimality and performance.

Key ideas. Our scheduler leverages an analytical model to estimate the Qernel execution time quickly. Then, based on the fidelity predictions from the estimator, it assigns and runs Qernels across space (which QPUs) and time (when). Last, our scheduler supports configurable policies for managing the aforementioned tradeoffs, prioritizing maximal fidelity, minimal waiting times, or a balanced approach. Our two main policies are a fast heuristic algorithm, similar to [62], and the first genetic-based multi-objective optimization algorithm.

Execution time estimation. To optimize for minimal waiting times, the scheduler must first estimate each Qernel's execution time and then aggregate the execution time estimations in each QPU's queue to compute the total waiting times. To estimate the execution time, we iterate the longest path (depth) of a Qernel (§ 4.2) that corresponds to the longest-duration gate chain and thus defines the Qernel's execution time. By summing the gate durations of each node in the longest path, we get the Qernel's total execution time.

Formula-based policy. Optimizing for conflicting objectives involves comparing two possible solutions (e.g., maximal fidelity vs. minimal waiting times). In the formula-based policy, we use the following scoring formula to compare and select between two possible assignments: $score = c\frac{f_2 - f_1}{f_1} - (1-c)\frac{t_2 - t_1}{t_1} + \beta\frac{u_2 - u_2}{u_1}$. This formula factors fidelity, waiting time, and utilization to determine which assignment is better, given priorities. The parameters are as follows: f_i: fidelity of the estimation result i, t_i: waiting time for the QPU from estimation result i, u_i: utilization of the QPU for estimation result i, $c \in (0,1)$: a system-defined constant that weighs the fidelity difference between estimations and finally, β: a system-defined constant acting as a weighting factor for utilization difference, balancing system throughput and fidelity. By selecting higher c, the system prioritizes fidelity over waiting times, and vice versa, and by selecting higher β the system prioritizes utilization over fidelity, and vice versa. By default, $c = \beta = 0.5$, which aims for balanced fidelity, waiting times, and utilization. Notably, prior work does not include utilization in the scheduling decisions [62].

Genetic algorithm policy. Genetic algorithms excel at optimizing for conflicting objectives by efficiently searching over vast search spaces, and for that, they can be used in the context of QOS. We formulate a multi-objective optimization problem with the conflicting objectives of fidelity vs. waiting times and use the NSGA-II genetic algorithm [17] to solve it. The algorithm creates a Pareto front of possible solutions (schedules), each achieving a different combination of average fidelity and average waiting times. Then, to select one of those schedules, we use the aforementioned formula to score each schedule and select the schedule with the highest score.

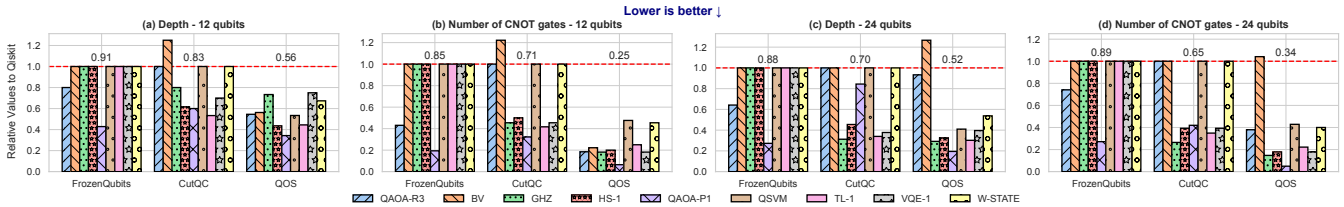


Figure 8: Error mitigator (§ 9.2). *Impact of the error mitigator on the circuit depth and the number of CNOTs. The circuits are optimized using budget $b=3$, and we compare against Qiskit (red horizontal line), FrozenQubits [4] and CutQC [73]. There is an average 46%, 38.6%, and 29.4% reduction in circuit depth, and 70.5%, 66% and 56.6% reduction in the number of CNOTs, respectively.*

9 Evaluation

9.1 Experimental Methodology

Experimental setup. We conduct two types of experiments: (1) classical tasks, such as circuit transpilation and trace-based simulations, and quantum tasks (2), which run on ***real*** QPUs for measuring the circuits' fidelities.

For (1), we use a server with a 64-core AMD EPYC 7713P processor and 512 GB ECC memory. For (2), we conduct our experiments on IBM Falcon r5.11 QPUs. Unless otherwise noted, we use the ***real*** Kolkata 27-qubit QPU hosted by IBM.

Framework and configuration. We use the *Qiskit* [33] Python SDK for compiling quantum circuits and running simulations. We compile quantum circuits with the highest optimization level (3) and run with 8192 shots. Each data point presented in the figures is the median of five runs.

Benchmarks. We study QOS on a set of circuits used in state-of-the-art NISQ algorithms, adopted from the 3 benchmark suits of Supermarq [76], MQT-Bench [60] and QASM-Bench [39]. The algorithms' circuits can be scaled by the number of qubits and depth. Specifically. We study 9 benchmarks: GHZ, W-State, Bernstein Vazirani (BV), Hamiltonian Simulation (HS-t), Quantum-enhanced Support Vector Machine (QSVM), Two Local Ansatz (TL-n), Variational Quantum Eigensolver (VQE-n), and Approximate Optimization Algorithm (QAOA-R/P), these benchmarks cover a wide range of relevant criteria for evaluating QOS.

For the TL and VQE circuits, we use circular and linear entanglement, respectively. The HS, VQE, and TL benchmarks are scalable by their circuit depth with the number of time-steps t and layers in the *ansatz* n. The QAOA-R/P circuits are initialized using regular/power-law graphs, respectively, with degree $d \in \{1,3\}$.

Metrics. We evaluate the following metrics: **(1) Hellinger Fidelity**. As defined in [21], where it ranges in $[0,1]$ and higher is better. **(2) Static Properties**. Number of *CNOT* gates and *depth* (§ 2.2). **(3) Utilization.** The effective QPU utilization (§ 7). **(4) Waiting Time**. The time a Qernel spends in a QPU's queue, waiting for execution, in seconds. **(5) Classical Overhead**. The error mitigation classical overheads as a factor of runtime increase (×). **(6) Quantum Overhead**. The error mitigation quantum overheads as a factor of numbers of circuits increase (×).

Baselines. We evaluate the error mitigator against the standalone fidelity-improving frameworks Qiskit v0.41, CutQC [73] and FrozenQubits [4]. We evaluate QOS's multi-programmer against [16], and we evaluate the synergy between the error mitigator, estimator, and multi-programmer vs. CutQC and [16] as a single baseline. Regarding QOS scheduler, to the best of your knowledge, [62] is the only peer-reviewed quantum scheduler, but it doesn't provide source code or enough technical details to faithfully implement it.

9.2 Error Mitigator

RQ1: *How well does the error mitigator improve the fidelity of circuits that run on NISQ QPUs?* We evaluate the performance of the error mitigator w.r.t the post-mitigation properties and fidelity of the circuits while also analyzing the classical and quantum costs of our approach.

Effect on the circuit depth and number of CNOTs. In Figure 8, we show the performance of the error mitigator on the circuits' depth and number of CNOTs, where we plot the relative difference in post-optimization circuit depth and the number of CNOTs between Qiskit (the red horizontal line) and FrozenQubits [4], CutQC [73], and our error mitigator. Figures 8 (a) and (c) show that the circuit depth decreases by 46%, 38.6%, and 29.4%, respectively. Figures 8 (b) and (d) show that the number of CNOTs decreases by 70.5%, 66%, and 56.6%, respectively. The improvement in both metrics against the baselines is attributed to the composability of our error mitigator, where the combined techniques achieve better results than any standalone technique. In cases where QOS under-performs compared to the baselines (e.g., BV benchmark for 24 qubits), it is because the error mitigator achieved the desired circuit size with the qubit reuse technique, which incurs additional costs (§5.1).

Impact on fidelity. Figure 9 shows the error-mitigated circuits' fidelity against Qiskit [58], CutQC [73], and FrozenQubits [4]. The results show a mean 2.6×, 1.6×, and 1.11× improvement for 12-qubit circuits, respectively, and a 456.5×, 7.6×, and 1.67× improvement for circuits of 24 qubits, respectively. The fidelity improvement is a consequence of lower circuit depths and fewer CNOTs, as shown in Figure 8.

Classical and quantum overheads. Figure 10 (a) shows the average classical and quantum overheads of the error

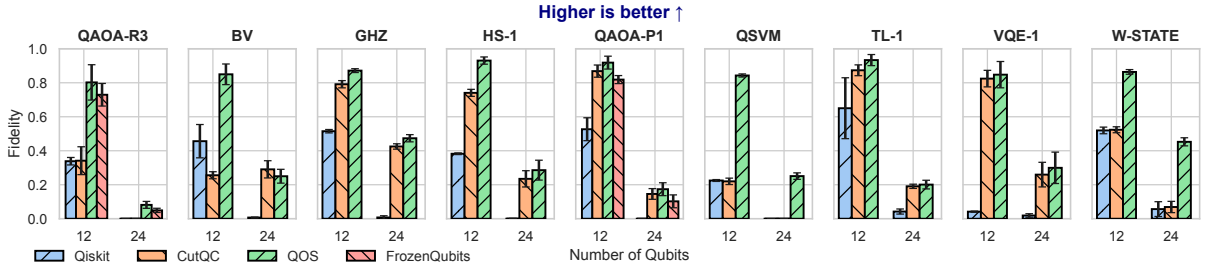


Figure 9: Error mitigator (§ 9.2). *Impact of the error mitigator on the circuit fidelity against Qiskit [58], CutQC [73], and FrozenQubits [4]. The circuits are optimized using budget $b=3$. There is a mean $2.6\times$, $1.6\times$, and $1.11\times$ improvement for 12-qubit circuits, respectively. There is a $456.5\times$, $7.6\times$, and $1.67\times$ improvement for circuits of 24 qubits, respectively.*

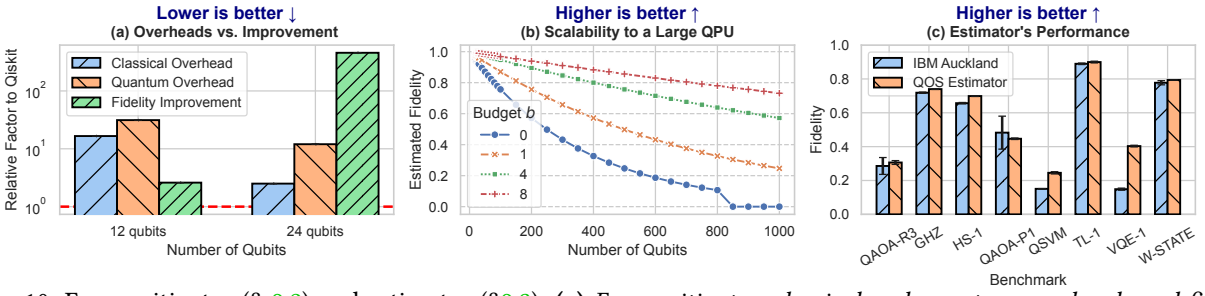


Figure 10: Error mitigator (§ 9.2) and estimator (§9.3). **(a)** *Error mitigator: classical and quantum overheads and fidelity improvement as a relative factor to Qiskit. For 24 qubits, the improvement outweighs the overheads.* **(b)** *Error mitigator: scalability to a large, hypothetical 1000-qubit QPU. Any budget $b>0$ achieves higher quality results than using no optimizations.* **(c)** *Estimator's performance: fidelity of IBM Auckland vs. the QPU automatically selected by the estimator.*

mitigator. The classical overhead is $16.6\times$ and $2.5\times$ for 12 and 24 qubits, respectively, and the quantum overhead is $31.3\times$ and $12\times$ for 12 and 24 qubits, respectively. However, fidelity improves by $2.6\times$ and $456.5\times$ for 12 and 24 qubits, respectively; therefore, for larger circuits, the fidelity improvement is worth the cost.

Scalability. To demonstrate that the error mitigator is scalable, we run the VQE-1 benchmark on a hypothetical 1000-qubit QPU with one-qubit gate errors of 10^{-4}, two-qubit gate errors of 10^{-3}, and measurement errors of 10^{-2}. We optimize with budget $b\in\{0,1,4,8\}$ and report the estimated fidelity. Figure 10 (b) shows that all budget b values improve the estimated fidelity, where higher b equals higher fidelity.

RQ1 takeaway. The error mitigator improves the properties of quantum circuits by 51% on average, leading to an improvement in fidelity of 2.6–$456.5\times$, while incurring justifiable classical and quantum overheads.

9.3 Estimator

RQ2: *How well does QOS's estimator address spatial and temporal heterogeneities?* We evaluate the estimator's precision in selecting the top-performing QPU for each benchmark. We establish a baseline using the on-average best-performing machine every calibration day. On the day of the experiment, IBM Auckland was the best-performing machine (also with the highest number of pending jobs).

Estimator's accuracy. Figure 10 (c) shows the fidelity of the eight benchmarks when run on QPUs selected by the estimator versus when run on the IBM Auckland QPU. The QPU selected for the BV benchmark is Auckland; therefore, we omit this result. For the rest of the benchmarks, the IBM Sherbrooke and Brisbane QPUs were automatically selected. Interestingly, the fidelity is on par or even higher than IBM Auckland, except for only one benchmark, the QAOA-P1, possibly because of its unique, power-law connectivity structure.

RQ2 takeaway. QOS's estimator automatically identifies QPUs with higher fidelity than the current standard practice.

9.4 Multi-programmer

RQ3: *How well does QOS's multi-programmer increase QPU utilization with minimum fidelity penalties?* We evaluate the impact of the multi-programmer on the fidelity of co-scheduled circuits for certain utilization thresholds.

Utilization vs. fidelity. Figure 11 (a) shows the average fidelity of nine benchmarks with utilization of 30%, 60%, and 88%. The three bars represent: no multi-programming *(No M/P)* refers to large circuits that run solo, baseline multi-programming *(Baseline M/P)* refers to [16], and QOS's multi-programming approach *(QOS M/P)*. There is an average $9.6\times$ improvement in fidelity compared to solo execution and an average 15% ($1.15\times$) improvement compared to the baseline.

Effective utilization. The results in Figure 11 (b) show that QOS achieves, on average, a 7.2% higher effective utilization (§ 7), with a maximum improvement of 10.1%. We attribute

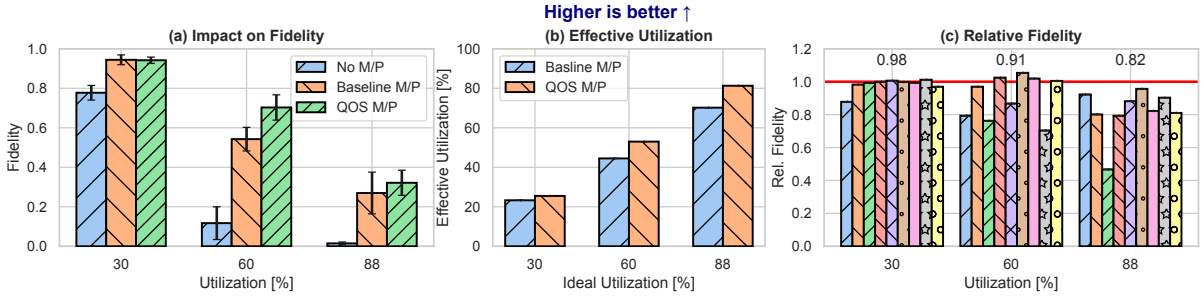


Figure 11: Multi-programmer (§ 9.4). **(a)** *Impact of multi-programming on fidelity. There is a 9.6× and 1.15× improvement compared to no multi-programming and the baseline, respectively.* **(b)** *Effective utilization. There is 7.2% higher effective utilization on average.* **(c)** *Relative fidelity w.r.t. solo circuit execution. There is an average 9.6% drop in fidelity due to QOS's multi-programming.*

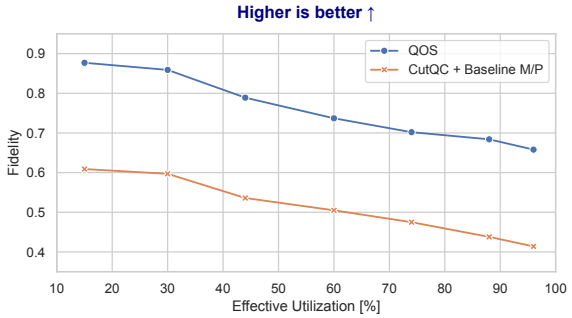


Figure 12: QOS error mitigator, estimator, and multi-programmer vs. CutQC [73] and the baseline multi-programming framework [16]. *QOS achieves* 48% *higher fidelity for the same utilization, on average.*

this improvement to the inclusion of temporal utilization.

Fidelity penalty vs. solo execution. In Figure 11 (c), we evaluate the fidelity penalty of multiprogramming vs. solo circuit execution for utilization of 30%, 60%, and 88%. The fidelity loss is 2%, 9%, and 18%, respectively. The average fidelity loss is 9.6% compared to solo execution, which is in line with previous studies [16, 40]. In the worst case (18%), the fidelity loss is caused by the restrictions in high-quality qubit allocations and the crosstalk errors.

RQ3 takeaway. The QOS multi-programmer improves fidelity by 1.15–9.6× and effective utilization by 7.2% compared to the baselines while incurring an acceptable fidelity penalty ($<10\%$) compared to solo execution.

9.5 QOS vs. Combined Baseline

RQ4: *What is the performance of the within- and cross-layer optimizations of QOS's error mitigator, estimator, and multi-programmer?* We evaluate the impact of QOS's components (without the scheduler) on the fidelity and utilization of the co-scheduled circuits for certain utilization thresholds.

Figure 12 shows the fidelity performance of QOS vs. CutQC [73] and the baseline multi-programming [16]. Here, we use a 27-qubit QPU, and a workload of all nine evaluation benchmarks with initial circuit sizes of 8-, 16-, and 24-qubits. We configure the error mitigator and CutQC to achieve a circuit size half of the original (i.e., 4, 8, and 12-qubit circuits). QOS achieves 48% higher fidelity than the baselines for the same utilization, on average. There are two main reasons for this: (1) the QOS components perform better individually as previously shown (§ 9.2, § 9.4) and (2) the synergy between the QOS components enables better (w.r.t. fidelity) co-scheduling decisions compared to standalone systems [16].

9.6 Scheduler

RQ5: *How well does QOS's scheduler balance fidelity vs. waiting times and balance the load across QPUs?* We evaluate our scheduler by generating a representative workload consisting of a dataset we collected during the development of QOS.

Dataset collection. During our exploration of the motivational challenges (§ 3) and experimentation and evaluation of the QOS components and their policies, we collected a dataset of 70.000 benchmark circuits and more than 7000 job runs in the quantum cloud. We use this dataset to simulate representative workloads, as we detail next.

Workload generation. To generate realistic workloads, we monitored all available QPUs on the IBM Quantum Cloud [32] for ten days in November 2023 to estimate the hourly job arrival rate. The average hourly rate is 1500 jobs per hour and is the baseline system workload for our evaluation.

Fidelity vs. waiting time. Figure 13 (a) shows the performance of the formula-based scheduling policy. We show the average fidelity and waiting time as the fidelity weight, c, changes (§ 8). A weight of 0.7 achieves $\sim$5× lower waiting times than full priority of fidelity while sacrificing only $\sim$2% fidelity. Figure 13 (b) shows the Pareto front of scheduling solutions generated by the genetic algorithm policy. A weight $c=0.5$ achieves 2× lower waiting times with 4% lower fidelity.

QPU load balancing. Figure 13 (c) shows the QPU load as the total runtime each QPU was active, in seconds, for the formula-based policy. All QPUs handle similar loads, with a maximum difference of 15.2%.

RQ5 takeaway. QOS scheduler balances the trade-off between waiting times and fidelity by reducing them 5× and only 2%, respectively while balancing the load across QPUs.

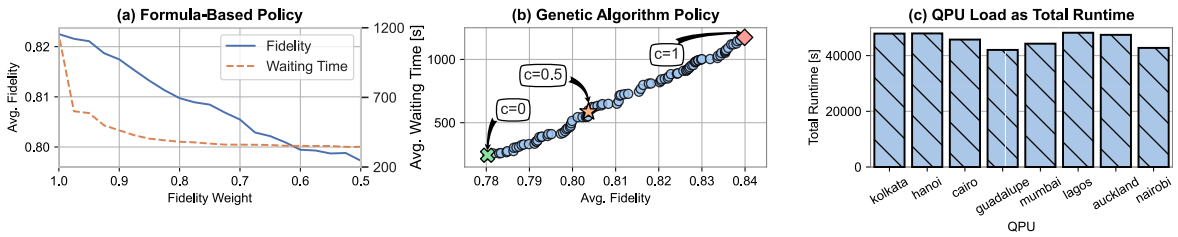


Figure 13: Scheduler (§ 9.6). **(a)** *Formula-based scheduling policy: Average fidelity vs. average waiting times. A fidelity weight* $c = 0.7$ *achieves* $\sim 5\times$ *lower waiting time for only* $\sim 2\%$ *lower fidelity.* **(b)** *Genetic algorithm policy: it creates a Pareto front of schedules, where a fidelity weight* $c = 0.5$ *achieves* $2\times$ *lower waiting times for* $\sim 4\%$ *fidelity decrease.* **(c)** QPU load as the total runtime of each QPU for the formula-based policy. The maximum load difference between any two QPUs is 15.2%.

10 Related work

Quantum error mitigation. Error mitigation techniques can be categorized as (1) execution pre- and post-processing [13–15, 42, 51–54, 71, 74, 75], (2) circuit divide-and-conquer [4, 6, 54, 73], and (3) qubit reuse [8, 29, 34, 48, 48, 49]. Unfortunately, all these techniques are implemented standalone without any infrastructure to compose them. Instead, QOS integrates and composes at least one technique per category in a single software stack, achieving higher fidelity and abstracting away the complexity from the programmer.

Quantum performance estimation. Prior work on quantum performance estimation has focused on resource estimation—predicting required physical qubits under fixed fidelity assumptions [7, 59]—and fidelity estimation using analytical models [47]. In contrast, QOS estimates fidelity under hardware constraints and supports both analytical and ML-based approaches.

Quantum multi-programming. Quantum multi-programming work [16, 40] focuses solely on mapping *two* circuits on a single QPU while providing similar fidelity to both (allocation fairness). However, they lack a systematic strategy for selecting compatible jobs to co-locate, and do not consider temporal utilization. Furthermore, neither work supports buffer zones to reduce crosstalk noise between co-located circuits. The delayed instruction scheduling optimization proposed by [16] is already available in the Qiskit transpiler via the ALAP scheduling pass [58], and thus is inherently used by QOS. Finally, [40] introduces a cross-program SWAP optimization technique, which tightly couples its multi-programming and scheduling logic; this makes it incompatible with any separate, system-level scheduler like QOS's, as relocating the bundled circuits to another QPU invalidates its core optimization technique.

Quantum job scheduling. Current quantum scheduling methods [62, 64, 72, 81] are limited because they (1) only perform single-to-many scheduling, (2) do not account for QPU utilization, (3) lack fine control over the fundamental design tradeoffs (§ 3.4), or (4) require manual input for final scheduling decisions. In contrast, QOS multiplexes circuits across space and time in a many-to-many fashion, increases QPU utilization, balances the inherent tradeoffs of quantum, and abstracts away the underlying resource management.

Cloud OSes, resource management, and job scheduling. Classical cloud operating, resource management, and job scheduling systems have been extensively researched in the past decades [28, 36, 37, 46, 56, 66, 67, 79, 80, 83, 84]. However, the classical domain, even when addressing accelerator heterogeneity, does not face the unique challenges of quantum computing (§ 3) and, as such, cannot be trivially adapted to accommodate its needs.

11 Conclusion and Discussion

We presented QOS, the first quantum operating system to (1) holistically address quantum computing challenges through a modular, policy-mechanism-separated architecture, (2) enable cross- and intra-layer optimizations via end-to-end co-design, (3) systematically explore key tradeoffs between fidelity, utilization, and waiting times, and (4) introduce novel abstractions and techniques across the stack, including the Qernel, composable error mitigation, compatibility-based multi-programming, and multi-objective scheduling.

QOS establishes foundational principles for quantum operating systems that remain relevant as QPU technology evolves. While hardware will improve, core challenges like noise, heterogeneity, and resource scarcity will persist, requiring tradeoffs between fidelity, utilization, and latency. These principles naturally extend to fault-tolerant quantum computing, where QOS's fidelity-aware scheduler and compatibility-based multi-programmer can guide logical qubit placement and error-corrected resource allocation.

Acknowledgements

We sincerely thank our shepherd and the anonymous reviewers for their feedback. We thank Karl Jansen and Stefan Kühn from the Center for Quantum Technology and Applications (CQTA)- Zeuthen for supporting this work by providing access to IBM quantum resources. We also thank Ahmed Darwish and Dmitry Lugovoy for their contributions to this work. Funded by the Bavarian State Ministry of Science and the Arts as part of the Munich Quantum Valley (MQV).

References

[1] Qiskit 600,000 registered users have run over 3 Trillion Circuit Executions. https://www.ibm.com/quantum/blog/quantum-software-vision. Accessed: 2025-05-05.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[3] AWS Bracket. https://aws.amazon.com/braket/. Accessed: 2025-05-05.

[4] Ramin Ayanzadeh, Narges Alavisamani, Poulami Das, and Moinuddin Qureshi. Frozenqubits: Boosting fidelity of qaoa by skipping hotspot nodes. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 311–324, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Azure Quantum. https://azure.microsoft.com/en-us/products/quantum. Accessed: 2025-05-05.

[6] Luciano Bello, Agata M. Brańczyk, Sergey Bravyi, Almudena Carrera Vazquez, Andrew Eddins, Daniel J. Egger, Bryce Fuller, Julien Gacon, James R. Garrison, Jennifer R. Glick, Tanvi P. Gujarati, Ikko Hamamura, Areeq I. Hasan, Takashi Imamichi, Caleb Johnson, Ieva Liepuoniute, Owen Lockwood, Mario Motta, C. D. Pemmaraju, Pedro Rivero, Max Rossmannek, Travis L. Scholten, Seetharami Seelam, Iskandar Sitdikov, Dharmashankar Subramanian, Wei Tang, and Stefan Woerner. Circuit Knitting Toolbox. https://github.com/Qiskit-Extensions/circuit-knitting-toolbox, 2023.

[7] Michael E. Beverland, Prakash Murali, Matthias Troyer, Krysta M. Svore, Torsten Hoefler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, and Alexander Vaschillo. Assessing requirements to scale to practical quantum advantage, 2022.

[8] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 286–300, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Sebastian Brandhofer, Ilia Polian, and Kevin Krsulich. Optimal partitioning of quantum circuits using gate cuts and wire cuts. *arXiv preprint arXiv:2308.09567*, 2023.

[10] Sergey Bravyi, Sarah Sheldon, Abhinav Kandala, David C. Mckay, and Jay M. Gambetta. Mitigating measurement errors in multiqubit experiments. *Phys. Rev. A*, 103:042605, Apr 2021.

[11] Lukas Brenner, Christophe Piveteau, and David Sutter. Optimal wire cutting with classical communication, 2023.

[12] Andrew J Daley, Immanuel Bloch, Christian Kokail, Stuart Flannigan, Natalie Pearson, Matthias Troyer, and Peter Zoller. Practical quantum advantage in quantum simulation. *Nature*, 607(7920):667–676, 2022.

[13] Siddharth Dangwal, Gokul Subramanian Ravi, Poulami Das, Kaitlin N. Smith, Jonathan Mark Baker, and Frederic T. Chong. Varsaw: Application-tailored measurement error mitigation for variational quantum algorithms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 362–377, New York, NY, USA, 2024. Association for Computing Machinery.

[14] Poulami Das, Swamit Tannu, Siddharth Dangwal, and Moinuddin Qureshi. Adapt: Mitigating idling errors in qubits via adaptive dynamical decoupling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 950–962, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Poulami Das, Swamit Tannu, and Moinuddin Qureshi. Jigsaw: Boosting fidelity of nisq programs via measurement subsetting. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 937–949, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Poulami Das, Swamit S. Tannu, Prashant J. Nair, and Moinuddin Qureshi. A case for multi-programming quantum computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 291–303, New York, NY, USA, 2019. Association for Computing Machinery.

[17] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[18] Matthew DeCross, Eli Chertkov, Megan Kohagen, and Michael Foss-Feig. Qubit-reuse compilation with mid-circuit measurement and reset. *Phys. Rev. X*, 13:041057, Dec 2023.

[19] Leonardo DiCarlo, Jerry M Chow, Jay M Gambetta, Lev S Bishop, Blake R Johnson, DI Schuster, J Majer, Alexandre Blais, Luigi Frunzio, SM Girvin, et al. Demonstration of two-qubit algorithms with a superconducting quantum processor. *Nature*, 460(7252):240–244, 2009.

[20] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014.

[21] Qiskit Hellinger Fidelity. https://docs.quantum.ibm.com/api/qiskit/0.31/qiskit.quantum_info.hellinger_fidelity. Accessed: 2025-05-05.

[22] I. M. Georgescu, S. Ashhab, and Franco Nori. Quantum simulation. *Rev. Mod. Phys.*, 86:153–185, Mar 2014.

[23] Quantum computer datasheet. https://quantumai.google/hardware/datasheet/weber.pdf. Accessed: 2025-05-05.

[24] Google Quantum Computing Service . https://quantumai.google/cirq/google/concepts. Accessed: 2025-05-05.

[25] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[26] Laszlo Gyongyosi and Sandor Imre. A survey on quantum computing technology. *Computer Science Review*, 31:51–71, 2019.

[27] David Hanneke, JP Home, John D Jost, Jason M Amini, Dietrich Leibfried, and David J Wineland. Realization of a programmable two-qubit quantum processor. *Nature Physics*, 6(1):13–16, 2010.

[28] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

[29] Fei Hua, Yuwei Jin, Yanhao Chen, Suhas Vittal, Kevin Krsulich, Lev S. Bishop, John Lapeyre, Ali Javadi-Abhari, and Eddy Z. Zhang. Caqr: A compiler-assisted approach for qubit reuse through dynamic circuit. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 59–71, New York, NY, USA, 2023. Association for Computing Machinery.

[30] H. Häffner, C.F. Roos, and R. Blatt. Quantum computing with trapped ions. *Physics Reports*, 469(4):155–203, 2008.

[31] IBM Quantum calibration jobs. https://docs.quantum.ibm.com/admin/calibration-jobs. Accessed: 2025-05-05.

[32] IBM Quantum. https://www.ibm.com/quantum-computing/. Accessed: 2025-05-05.

[33] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with qiskit, 2024.

[34] Hanru Jiang. Qubit recycling revisited. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[35] P. V. Klimov, J. Kelly, Z. Chen, M. Neeley, A. Megrant, B. Burkett, R. Barends, K. Arya, B. Chiaro, Yu Chen, A. Dunsworth, A. Fowler, B. Foxen, C. Gidney, M. Giustina, R. Graff, T. Huang, E. Jeffrey, Erik Lucero, J. Y. Mutus, O. Naaman, C. Neill, C. Quintana, P. Roushan, Daniel Sank, A. Vainsencher, J. Wenner, T. C. White, S. Boixo, R. Babbush, V. N. Smelyanskiy, H. Neven, and John M. Martinis. Fluctuations of energy-relaxation times in superconducting qubits. *Phys. Rev. Lett.*, 121:090502, Aug 2018.

[36] Kubernetes. https://kubernetes.io/, 2024.

[37] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8343–8354. Curran Associates, Inc., 2020.

[38] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2), February 2023.

[39] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2), feb 2023.

[40] Lei Liu and Xinglei Dou. Qucloud: A new qubit mapping mechanism for multi-programming quantum computing in cloud environment. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 167–178, Feb 2021.

[41] Xiao-Song Ma, Thomas Herbst, Thomas Scheidl, Daqing Wang, Sebastian Kropatschek, William Naylor, Bernhard Wittmann, Alexandra Mech, Johannes Kofler, Elena Anisimova, et al. Quantum teleportation over 143 kilometres using active feed-forward. *Nature*, 489(7415):269–273, 2012.

[42] Filip B Maciejewski, Zoltán Zimborás, and Michał Oszmaniec. Mitigation of readout noise in near-term quantum devices by classical post-processing based on detector tomography. *Quantum*, 4:257, 2020.

[43] Filip B. Maciejewski, Zoltán Zimborás, and Michał Oszmaniec. Mitigation of readout noise in near-term quantum devices by classical post-processing based on detector tomography. *Quantum*, 4:257, April 2020.

[44] Kosuke Mitarai and Keisuke Fujii. Constructing a virtual two-qubit gate by sampling single-qubit operations. *New Journal of Physics*, 23(2):023021, feb 2021.

[45] Prakash Murali, David C. Mckay, Margaret Martonosi, and Ali Javadi-Abhari. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1001–1016, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[47] Paul Nation, Matthew Treinish, and Clemens Possel. mapomatic. https://github.com/Qiskit-Partners/mapomatic, 2022.

[48] Alexandru Paler, Robert Wille, and Simon J. Devitt. Wire recycling for quantum circuit optimization. *Phys. Rev. A*, 94:042337, Oct 2016.

[49] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. Unqomp: Synthesizing uncomputation in quantum circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 222–236, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Tirthak Patel, Abhay Potharaju, Baolin Li, Rohan Basu Roy, and Devesh Tiwari. Experimental evaluation of nisq quantum computers: Error measurement, characterization, and implications. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, Nov 2020.

[51] Tirthak Patel and Devesh Tiwari. Disq: A novel quantum output state classification method on ibm quantum computers using openpulse. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Tirthak Patel and Devesh Tiwari. Veritas: Accurately estimating the correct output on noisy intermediate-scale quantum computers. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, Nov 2020.

[53] Tirthak Patel and Devesh Tiwari. Qraft: Reverse your quantum circuit and know the correct program output. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 443–455, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Tirthak Patel, Ed Younis, Costin Iancu, Wibe de Jong, and Devesh Tiwari. Quest: systematically approximating quantum circuits for higher output fidelity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 514–528, New York, NY, USA, 2022. Association for Computing Machinery.

[55] Tianyi Peng, Aram W. Harrow, Maris Ozols, and Xiaodi Wu. Simulating large quantum circuits on a small quantum computer. *Phys. Rev. Lett.*, 125:150504, Oct 2020.

[56] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[57] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, aug 2018.

[58] Qiskit Transpiler. https://qiskit.org/documentation/apidoc/transpiler.html. Accessed: 2025-05-05.

[59] N. Quetschlich, L. Burgholzer, and R. Wille. MQT Predictor: Automatic Device Selection with Device-Specific Circuit Compilation for Quantum Computing. *ACM Transactions on Quantum Computing (TQC)*, 2024.

[60] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum*, 7:1062, July 2023.

[61] Gokul Subramanian Ravi, Kaitlin N. Smith, Pranav Gokhale, and Frederic T. Chong. Quantum computing in the cloud: Analyzing job and machine characteristics. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 39–50, 2021.

[62] Gokul Subramanian Ravi, Kaitlin N. Smith, Prakash Murali, and Frederic T. Chong. Adaptive job and resource management for the growing quantum cloud. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 301–312, Oct 2021.

[63] Diego Ristè, Marcus P Da Silva, Colm A Ryan, Andrew W Cross, Antonio D Córcoles, John A Smolin, Jay M Gambetta, Jerry M Chow, and Blake R Johnson. Demonstration of quantum advantage in machine learning. *npj Quantum Information*, 3(1):16, 2017.

[64] Marie Salm, Johanna Barzen, Frank Leymann, and Benjamin Weder. Prioritization of compiled quantum circuits for different quantum computers. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1258–1265, March 2022.

[65] Schrödinger's cat. https://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s_cat. Accessed: 2025-05-05.

[66] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, New York, NY, USA, 2013. Association for Computing Machinery.

[67] Omar Sefraoui, Mohammed Aissaoui, Mohsine Eleuldj, et al. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[68] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[69] Irfan Siddiqi. Engineering high-coherence superconducting qubits. *Nature Reviews Materials*, 6(10):875–891, 2021.

[70] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintao Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 113–125, New York, NY, USA, 2018. Association for Computing Machinery.

[71] Kaitlin N. Smith, Gokul Subramanian Ravi, Prakash Murali, Jonathan M. Baker, Nathan Earnest, Ali Javadi-Cabhari, and Frederic T. Chong. Timestitch: Exploiting slack to mitigate decoherence in quantum circuits. *ACM Transactions on Quantum Computing*, 4(1), oct 2022.

[72] Samuel Stein, Nathan Wiebe, Yufei Ding, Peng Bo, Karol Kowalski, Nathan Baker, James Ang, and Ang Li. Eqc: Ensembled quantum computing for variational quantum algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 59–71, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. Cutqc: using small quantum computers for large quantum circuit evaluations. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 473–486, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Swamit S. Tannu and Moinuddin Qureshi. Ensemble of diverse mappings: Improving reliability of quantum computers by orchestrating dissimilar mistakes. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 253–265, New York, NY, USA, 2019. Association for Computing Machinery.

[75] Swamit S. Tannu and Moinuddin K. Qureshi. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 279–290, New York, NY, USA, 2019. Association for Computing Machinery.

[76] Teague Tomesh, Pranav Gokhale, Victory Omole, Gokul Subramanian Ravi, Kaitlin N. Smith, Joshua Viszlai, Xin-Chuan Wu, Nikos Hardavellas, Margaret R. Martonosi, and Frederic T. Chong. Supermarq: A scalable quantum benchmark suite. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 587–603, 2022.

[77] Caroline Tornow, Naoki Kanazawa, William E. Shanks, and Daniel J. Egger. Minimum quantum run-time characterization and calibration via restless measurements with dynamic repetition rates. *Phys. Rev. Appl.*, 17:064061, Jun 2022.

[78] Nathaniel Tornow, Emmanouil Giortamis, and Pramod Bhatotia. Scaling quantum computations via gate virtualization. *Proceedings of the ACM on Programming Languages*, 9(PLDI), June 2025. Accepted: 2025-03-06; Received: 2024-11-14.

[79] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[80] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[81] Meng Wang, Poulami Das, and Prashant J. Nair. Qoncord: A multi-device job scheduling framework for variational quantum algorithms. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 735–749, 2024.

[82] Nicolas Wittler, Federico Roy, Kevin Pack, Max Werninghaus, Anurag Saha Roy, Daniel J. Egger, Stefan Filipp, Frank K. Wilhelm, and Shai Machnes. Integrated tool set for control, calibration, and characterization of quantum devices applied to superconducting qubits. *Phys. Rev. Appl.*, 15:034080, Mar 2021.

[83] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[84] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.



Scalio: Scaling up DPU-based JBOF Key-value Store with NVMe-oF Target Offload

Xun Sun[*1], Mingxing Zhang[*1,2], Yingdi Shan[*], Kang Chen[*], Jinlei Jiang[*], Yongwei Wu[*†2]
[*]*Tsinghua University,* [†]*Quan Cheng Laboratory*

Abstract

The rapid growth of data-intensive applications has created a demand for high-density storage systems. Data-Processing-Unit-based (DPU-based) Just a Bunch of Flash (JBOF) solutions provide an energy-efficient and cost-effective architecture to meet this need. However, existing JBOF solutions struggle with scalability when handling an increasing number of attached SSDs, due to their heavy reliance on the DPU's CPU for SSD I/O operations.

In this paper, we introduce Scalio, a scalable disaggregated key-value store designed to address the limitations of current DPU-based JBOF systems. Scalio offloads as many SSD I/O operations as possible to the DPU's network I/O capabilities, including traditional RDMA verbs and a recent hardware optimization, NVMe over Fabrics Target Offload. Additionally, Scalio incorporates a two-layer design with compact in-memory data structures to handle hot read traffic and manage bursty writes. One of the key challenges in this design is ensuring consistency between the DRAM states in the DPU and the SSD states, which, unlike CPU L1/L2 caches, are not automatically synchronized through hardware cache coherence protocols. To address this, Scalio introduces an RDMA-based cache consistency protocol that guarantees linearizability across the system, despite the disaggregated nature of the architecture.

Our experiments show that Scalio significantly improves both scalability and throughput, achieving up to 3.3× higher throughput compared to existing systems, especially in high-density SSD configurations.

1 Introduction

1.1 Motivation

The demand for high-density storage solutions has rapidly increased with the growth of data-intensive applications [8, 21, 27, 30, 36]. To address this need, Data-Processing-Unit-based (DPU-based) Just a Bunch of Flash (JBOF) storage systems (Figure 1) have emerged as cost-effective and energy-efficient solutions, drawing significant attention from both academia [12, 13, 15, 16, 23, 24, 35, 55, 56, 62] and industry [6, 33, 34, 38, 39, 42, 43]. As they are designed specifically for network and storage acceleration, DPUs typically consume far less power and cost compared to traditional storage servers [4, 5, 48]. For instance, even high-end DPUs like NVIDIA's BlueField-3 use only 75–150 watts [44], significantly less than a typical Xeon CPU, which consumes 300–500 watts [19].

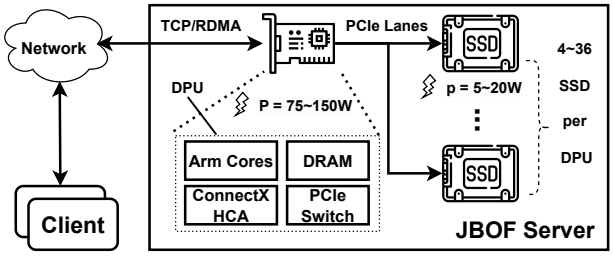


Figure 1: DPU-based JBOF systems' architecture [6, 33].

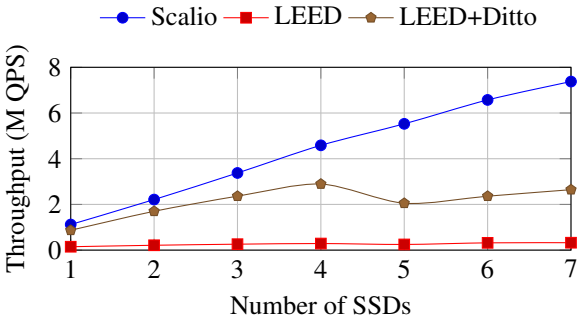


Figure 2: The throughput of Scalio with different number of SSDs attached to each DPU under YCSB [11] workload C. Scalio is compared to: 1) SOTA JBOF SSD key-value store LEED [16]; and 2) LEED combined with SOTA disaggregated DRAM key-value store Ditto [52] as its in-memory cache.

To maximize these benefits, increasing storage density by connecting more SSDs per DPU is a promising strategy. Theoretically, if each SSD offers a throughput of t and consumes power p (typically 5–20W [47, 63]), and the control plane (mainly the DPU) consumes power P, then with n SSDs, the throughput per watt is $\frac{nt}{P+np}$. Since P is typically more than ten times larger than p, the potential maximum throughput per watt increases sharply as the number of SSDs per DPU grows. This vision is further reinforced by the recent emergence of

[1]Equal contribution
[2]Corresponding authors: zhang_mingxing@mail.tsinghua.edu.cn, wuyw@tsinghua.edu.cn

DPU-based JBOF servers from vendors like SuperMicro [33] (a 36-SSD JBOF) and AIC [1, 51] (a 26-SSD JBOF), indicating an urgent need for optimizations targeting scenario of high-density storage with energy-efficient cores.

However, current software stacks struggle to **scale up** effectively with higher SSD counts in JBOF systems. This issue is particularly evident in small key-value workloads, which are common in latency- and throughput-critical applications like social networks and e-commerce. Serving these scenarios with high-density JBOF platforms should ideally saturate the aggregated IOPS of the large bunch of SSDs, providing high throughput with low latency while being cost- and energy-efficient. Yet, as shown in Figure 2, the throughput of existing disaggregated JBOF key-value store systems like LEED [16] plateaus when more than four SSDs are connected per DPU, highlighting the need for optimization of small key-value workloads to fully exploit the storage potential of JBOF systems. Moreover, this focus aligns with the increasing demand for low-latency, small operations in cloud-native applications, which dominate modern distributed storage needs [3, 9, 11, 32, 53, 59]. By optimizing for small key-value workloads, Scalio not only enhances performance but also improves scalability in high-density, energy-efficient storage systems.

1.2 Our Contributions

Our analysis attributes the above scale-up limitation of JBOF key-value stores to its heavy reliance on the DPU's CPU for SSD I/O operations, which derives from the micro-benchmark results in § 2.1. While the small ARM cores in DPUs offer energy efficiency, they lack the processing power to handle the combined IOPS of multiple SSDs, making the DPU's CPU a critical bottleneck in high-density configurations [12, 25].

Interestingly, unlike the constrained CPU resources, we observe that network resources are largely underutilized in these IOPS-bound scenarios. Across all our experimental settings, less than 1% of the maximum network IOPS is utilized, which exists because there is a three-order-of-magnitude gap between the performance of SSD I/O operations and the DPU's RDMA read/write capabilities [18, 45].

Based on this observation, we aim to offload as many SSD I/O operations as possible to the network. We leverage a technique called NVMe over Fabrics (NVMe-oF) Target Offload, which implements the NVMe-oF standard directly on the server-side DPU's Host Channel Adapter (HCA) [14, 28, 58]. With this approach, clients can send NVMe commands directly to server-side SSDs over network, utilizing the network IOPS resources without burdening the DPU's CPU. The DPU's HCA processes all regular I/O requests using hardware logic, communicating directly with NVMe PCI devices via peer-to-peer PCI communications [40, 41]. As illustrated in Figure 3, this setup effectively bypasses the DPU's CPU. More details will be provided in §2.2.

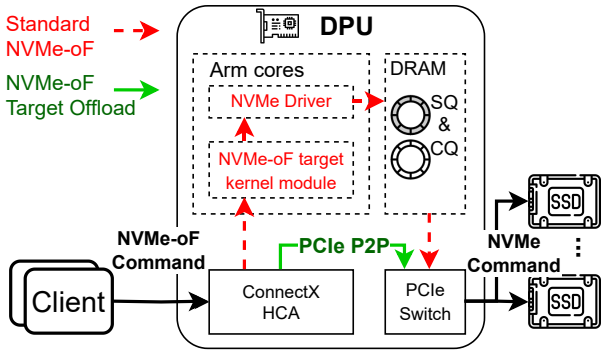


Figure 3: The data flow of the standard NVMe-oF and NVMe-oF Target Offload technology.

Moreover, our further experiments reveal that even when all SSD read operations are offloaded via NVMe-oF target offload, the DPU's CPU can still become a bottleneck in write-intensive workloads. This is primarily because CPU resources are heavily consumed by in-memory concurrency control mechanisms and handling random SSD writes. To overcome this challenge, we develop a **two-layer design** that employs succinct data structures residing in the DPU's DRAM. This design absorbs recent hot reads through offloaded RDMA reads and manages bursty writes via buffered group commits (§4.2). The in-memory data structure is shared between the DPU's CPU and clients' remote RDMA operations, allowing part of its maintenance overhead to be offloaded to clients.

However, while RDMA and NVMe-oF target offload reduce CPU burden by bypassing it, they introduce new consistency challenges. In a disaggregated architecture, clients operate independently without direct interconnection or synchronization. When read/write logic is partially or fully offloaded to clients' remote operations, concurrent accesses from multiple clients can corrupt data if there are no proper mechanisms to control access to shared resources.

The crux of the problem is that, unlike L1/L2 caches in CPUs, which are automatically maintained via hardware cache coherence protocols [20], **the DRAM states (e.g., indexes and caches for hot key-value pairs) residing in the DPU are not automatically consistent with the SSD states**. With NVMe-oF target offload, clients read data directly from the SSD, bypassing the DPU's CPU and DRAM. This necessitates explicit cache invalidation mechanisms initiated by remote clients. Without proper handling, this can lead to cache consistency issues that violate linearizability guarantees [46]. In § 5, we delve deeper into this problem and introduce our RDMA-driven cache consistency protocol.

This protocol effectively manages the flushing of buffered data and cache invalidations to maintain consistency and ensure linearizability. It also carefully partitions the roles between clients' offloaded operations and the DPU's server-

side operations, allowing us to reduce the excessive use of heavy RDMA Compare-and-Swap (CAS) operations for synchronization. This is crucial for performance as studies [7,10,22,31,57,61] have shown that the maximum IOPS of RDMA CAS is ten times less than that of RDMA read/write.

In this paper, we propose Scalio, the first disaggregated key-value store leveraging NVMe-oF target offload to achieve scalability in the DPU-based JBOF architecture. We evaluated Scalio against two baseline systems—LEED [16] and LEED with Ditto [52] as the in-memory cache—using YCSB [11] workloads A, B, C, D, and F. Through experiments varying the number of SSDs from 1 to 7, we demonstrated Scalio's scalability and verified its performance improvements. These improvements are a result of several key mechanisms: in-memory caching, offloaded reads, and batched writes, which reduce CPU usage and optimize SSD IOPS utilization. Sensitivity analysis further confirmed that Scalio's performance is robust to changes in key parameters. Our experiments show that Scalio achieves a 1.8× to 3.3× speedup over LEED with Ditto, and 2.5× to 17× over LEED, highlighting the solid performance improvements of Scalio.

2 Background and Motivation

2.1 JBOF based Key-value Store

DPU-based Just a Bunch of Flash (JBOF) architectures represent a promising evolution in storage systems, offering significant advantages in energy efficiency compared to traditional CPU-based storage servers [12, 16, 35, 55, 62]. Typically, a JBOF system combines multiple NVMe SSDs with one or two Data Processing Unit (DPU) into a compact 1U/2U high-density chassis [6, 33]. Rather than relying on the general-purpose CPUs and NICs of conventional systems, The DPU consolidates networking, data acceleration, and storage processing into a single, specialized ASIC (Application-Specific Integrated Circuit), providing both high performance and reduced energy consumption [5].

The energy efficiency and high-density storage capabilities of JBOF architecture make it an ideal platform for key-value storage systems, especially in environments requiring high performance and energy savings [27, 62]. As an illustration, LEED [16] is a key-value store specialized for the JBOF platform. It uses a log-based data structure on the SSDs for reducing random I/O and a small, hash-based index structure in memory for indexing. According to its experiements with one 8-core Broadcom Stingray PS1100R SmartNIC [5] and four Samsung DCT983 SSDs [50], LEED achieves near-full saturation of the SSD I/O capabilities.

However, we observed that CPU resourses remain a primary bottleneck in LEED. It allocates one core for control-plane tasks, three cores for RDMA requests, and four cores (i.e., one dedicated core for every SSD) to handle SSD I/O operations, **which inherently limits the scalability of LEED**.

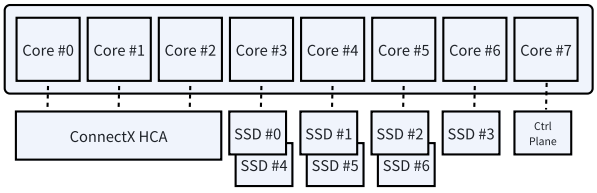


Figure 4: CPU allocation of the testbed.

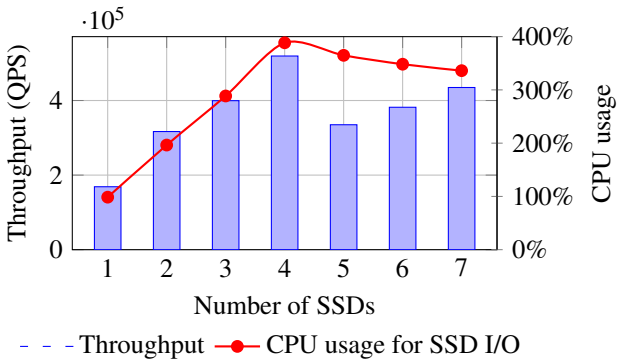


Figure 5: LEED's throughput and CPU usage for handling SSD I/O operations as the number of SSDs scale up.

In order to evaluate the system as the number of SSDs increases, we augment the SSD binding strategy by allocating the SSDs to the CPU cores in a round-robin approach, as in Figure 4. We vary the number of SSDs from 1 to 7, and plot the system throughput along with the CPU usage in handling SSD I/Os in YCSB C workload that exhibits a uniform request distribution, as in Figure 5.

As can be seen from the experiment result, the system throughput reaches a maximum when using 4 SSDs, whereas the CPU usage also approaches the limit (400% for handling SSD I/O in LEED's configuration)[1]. As the number of SSDs increases, the CPU usage gets saturated, causing the throughput unable to scale up.

Interestingly, we also measured LEED's network I/O utilization, and the results show that LEED's network I/O utilization remains less than 1%. In our tests, LEED achieved no more than 600K IOPS over the network, while the Mellanox ConnectX-6 Host Channel Adapter it uses is capable of handling up to 200M IOPS [45]. This underutilization of network I/O capabilities offers an opportunity for achieving better performance in a high-density setup, by offloading calculation from the DPU's CPU to the HCA and releasing burden on the DPU's CPU.

[1]We use perf to quantify the CPU usage of LEED in handling SSD I/Os, which derives from calling spdk_bdev_*, kv_* (functions of LEED kv store) and their corresponding callback functions.

2.2 NVMe over Fabrics Target Offload

NVMe over Fabrics (NVMe-oF) [12, 17, 37] extends the Non-Volatile Memory Express (NVMe) standard, which was originally designed for fast storage access over PCIe, to networked environments. It enables high-speed data transfer between host and remote NVMe devices across data center networks.

In standard CPU-based NVMe-oF implementations, the host system encapsulates NVMe commands into NVMe-oF requests and sends them over the network to the target storage device. The target server's CPU handles these requests through a kernel module or a userspace target process (typically implemented via SPDK [60]). These modules transform the incoming requests into block I/O structures, issues operations to the NVMe device, and replies to the client. Processing these I/O requests consumes significant CPU cycles on the target, especially when the IOPS is high.

With the increasing adoption of NVMe-oF in data centers, modern DPUs have introduced a specialized hardware optimization called NVMe-oF Target Offload [14, 28, 58]. As illustrated in Figure 3, this implementation delegates the handling of NVMe operations and SSD commands entirely to the Host Channel Adapter (HCA) hardware via PCIe P2P commands. By offloading these tasks, the CPU usage on the target side is minimized. This allows the system to scale more effectively, handling higher IOPS with lower latency without being limited by the CPU's processing power.

To demonstrate the benefits of NVMe-oF target offload, we conducted performance measurements comparing standard NVMe-oF and NVMe-oF target offload implementations. Using `fio` we measured the maximum read IOPS of both implementations as the number of SSDs increased. The results are presented in Figure 6.

Our experiments show that NVMe-oF target offload achieves similar IOPS performance to standard NVMe-oF implementations. In addition, NVMe-oF target offload offers great CPU savings. Compared with the standard NVMe-oF implementation that consumes 562% target CPU, the target offload implementation consumes 0 target CPU resource. Therefore, NVMe-oF target offload offers significant advantages for implementing scalable disaggregated storage systems. However, adopting NVMe-oF target offload requires careful coordination between the host and target to maintain data consistency and ensure efficient communication.

3 Overview

In our analysis of existing DPU-based JBOF disaggregated key-value stores, we identified excessive CPU consumption on the DPU as a key scalability limitation. To overcome this challenge, we redesigned the in-memory data structures and optimized the read and write workflows. Our approach aims to offload as many I/O operations as possible to clients through one-sided RDMA/NVMe-oF network operations while ensuring linearizability and consistency are maintained. Figure 7 compares the workflow of the current state-of-the-art system LEED with the workflow of our proposed solution.

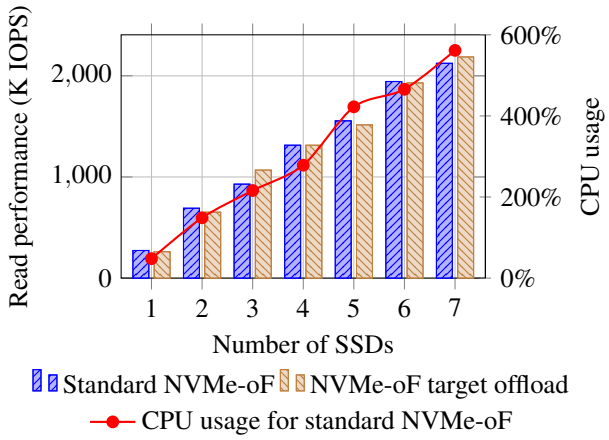


Figure 6: The read IOPS performance of the standard NVMe-oF implementation and the NVMe-oF target offload implementation under fio tests, along with the CPU usage of the standard NVMe-oF implementation as the number of SSDs scale up. The CPU usage of the NVMe-oF target offload implementation is zero.

3.1 Read Workflow

Existing systems, like LEED, rely on two-sided RDMA-based RPC calls for both read and write operations. In these systems, the DPU handles all queries and updates to the in-memory index along with data reads and writes from the SSD, resulting in high CPU usage on the DPU.

In contrast, Scalio, as shown in Figure 7 (b), introduce a multi-layer in-memory structure that integrates the SSD index with an in-memory cache. This allows direct RDMA-based queries and updates. When a client requests a key, the operation proceeds in two phases:

1. In-Memory Query Phase (Steps 1-3): The client uses RDMA read operations to fetch the hash block matching the hash value of the queried key. Each hash block contains multiple slots to hold several key-value pairs and a linear search is performed by the client to find the desired key. If the key is found in the block (cache hit), the value is returned immediately. Otherwise, a victim slot will be determined based on the oldest last_ts (modified in the background by clients finishing cache operations), and the client will evict the outdated data and lock the slot with an RDMA CAS operation on the complete field. Validation with a double-read is required during the process of locking in order to ensure atomicity, which will be further explained in § 5.2.

2. SSD Access and Cache Update Phase (Steps 4-5): If the key is not found in the cache (cache miss), the in-memory

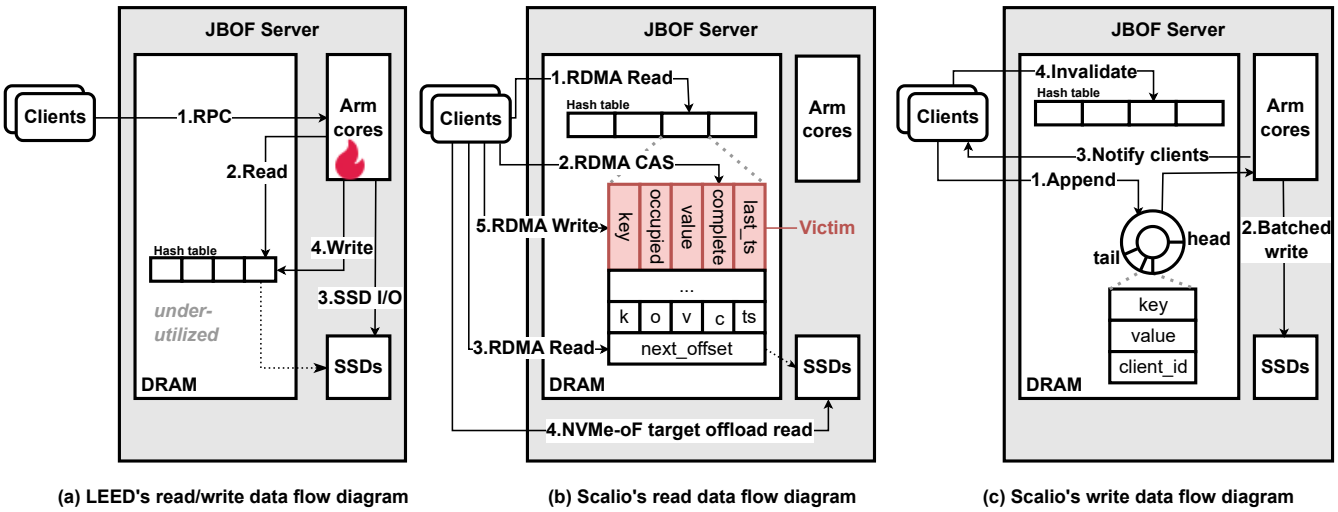


Figure 7: The workflows comparision between LEED's and Scalio.

structure provides the SSD location with the next_offset field. The client then performs a one-sided NVMe-oF target offload operation to read the data from the SSD. After retrieving the data, the same client updates the victim slot with RDMA write operations and frees the lock.

All of these operations fully utilize the network I/O capabilities with one-sided network primitives and hence bypass the DPU's CPU. To implement this workflow both efficiently and correctly, we face two main challenges that will be addressed in subsequent sections. First, we observe that RDMA network bandwidth remains under-utilized, particularly with small key-value pair workloads. To optimize this, we configure the capacity of each hash block to a larger value, accommodating more key-value pairs with the same key hash without compromising the network IOPS. Second, to handle concurrent updates to the in-memory structure by multiple clients and the DPU's CPU, we introduce two additional flags (occupied and complete) for each in-memory item and a cache consistency protocol to maintain linearizability (discussed in §5).

3.2 Write Workflow

Unlike the read workflow, which is entirely one-sided, the write workflow of Scalio involves both RDMA operations and processing by the DPU's CPU. In the original LEED implementation, each write involves two SSD write operations performed by the DPU's CPU, creating a bottleneck in write-heavy workloads. To address this, we implement a server-side batching method that enables group commits, reducing the number of SSD writes. The process involves:

1. Appending Request Data (Step 1): Clients append their write requests to an in-memory ring buffer on the DPU using RDMA operations.

2. Batched Writing (Step 2): The DPU's CPU continuously polls the ring buffer, flushes the write requests to the SSDs in batches and updates the next_offset field.

3. Client Notification (Step 3): After the updates are written to the SSDs and the new offsets are updated in-memory, the DPU's CPU notifies the clients of the successful write.

4. Cache Invalidation (Step 4): Clients invalidate outdated cache entries using RDMA write operations.

During the buffering period, the client's write operation is held until the DPU's CPU processes the batch, updates the next_offset and sends a notification. The write is considered committed when the corresponding next_offset field is updated and any oudated item is invalidated from the cache. More details are provided in §4.2.

By incorporating these optimizations, we reduce the CPU load on the DPU and enable the system to scale effectively as the number of SSDs increases. Our approach addresses the scalability bottlenecks of current systems, fully leveraging the capabilities of DPU-based JBOF architectures.

4 Scalio Implementation

In this section, we present the implementation details of Scalio, focusing on its mechanisms for efficient query and update operations. Scalio's design centers around two key aspects: (1) an in-memory index shared between the DPU and clients, which serves as cache and provides a consistent and synchronized view of system data; and (2) a batched write mechanism, leveraging group commits to improve write throughput while handling bursty workloads.

4.1 Index Structure

The index structure of Scalio includes an RDMA-accessible hash table for caching frequently accessed key-value pairs.

This design minimizes the need for DPU's CPU involvement by enabling clients to interact directly with the cache.

The hash table hashes keys to consecutively addressed blocks, each containing multiple slots for storing key-value pairs, as shown in Figure 7 (b). Given Scalio's focus on small key-value pairs, both the key and value are stored inline within each slot. Our prior observations suggest that network I/O capabilities have significant headroom for further utilization, allowing us to increase block size without saturating the network. Based on sensitivity analysis in § 6.5, we found that block sizes up to 1KB can be used without overloading the network bandwidth. This size is sufficient to cache small key-value pairs, with each block holding up to 10 key-value pairs of 100 bytes each, or more for smaller pairs. This design enables clients to fetch an entire block in a single round trip, reducing query latency for frequently accessed items by allowing quick access to the desired data.

Apart from the key and value, each slot in the hash table contains three additional fields: occupied, complete, and last_ts. The first two fields serve as flags to indicate the cache consistency state of the slot, as discussed in § 5. The last_ts field records the timestamp of the most recent access to the k-v pair, and victim slots are determined by the oldest last_ts.

When a client accesses a key-value pair, it updates the corresponding last_ts field via an RDMA write operation in the background. The use of timestamps for LRU eviction in disaggregated memory systems has been evaluated in previous works [52] and is shown to offer sufficient performance.

Each block in the hash table contains an offset to the SSD index, which points to the corresponding hash-based index region in the SSD that indexes key-value pairs with the same hash. The implementation of such data structure in the SSD has been thoroughly discussed in prior works such as LEED [16]. When a client fails to find the desired key in the cache, it will use the offset to initiate NVMe-oF target offload operations, retrieving the data directly from the SSD.

4.2 Batched Write Mechanism

The batched write mechanism in Scalio consists of three main components: (1) a ring buffer for temporarily storing key-value updates with client IDs, (2) a batched write process for efficiently flushing updates to SSDs, and (3) a group commit mechanism to ensure the atomicity of batched operations.

The JBOF server handles incoming updates submitted by clients via RDMA send operations and appends these updates to the ring buffer, as shown in Figure 7 (c). It then polls it in a batched manner, accumulating updates for bulk flushing to SSDs. After writing to the SSDs, the server updates the corresponding next_offset field and then notifies clients of completion using their recorded client IDs. This batched write mechanism improves throughput by absorbing bursty write requests and reducing the DPU's CPU load through fewer SSD I/O operations.

A group commit process is triggered when the JBOF server accumulates a batch. The DPU flushes the batch to SSDs in a single deterministic operation (as long as the batch size is smaller than the SSD block granularity, such as 4KB). Afterward, the DPU updates the next_offset field in the in-memory data structures with the new SSD offsets, reflecting the locations of the written data. The group commit is completed when the DPU notifies all relevant clients that their updates have been successfully written. Scalio only acknowledges the write operations to the client when the update is successfully logged onto the SSDs, so that consistency is guaranteed in the event of server crashing.

While these mechanisms improve throughput and reduce the DPU's CPU load, determining the precise linearization point of write operations remains a challenge. Specifically, maintaining consistency between cached data and the underlying SSD storage is crucial, especially in a disaggregated system where client-server interactions may occur asynchronously. This challenge will be addressed in § 5, where we explain how Scalio ensures correctness while optimizing performance through its cache consistency protocol.

5 Cache Consistency Protocol

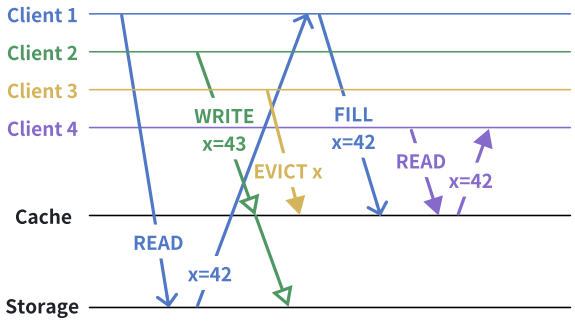
In a disaggregated architecture, certain operations are offloaded to clients, allowing them to bypass the JBOF server and directly interact with the underlying storage. This design improves performance by reducing the DPU's workload but also creates a critical challenge: ensuring that the cached data in the in-memory data structures remains consistent with the actual data stored on the SSDs. Naive cache consistency protocols fail to ensure linearizability in the disaggregated architecture, which will be explained in § 5.1. Therefore, maintaining cache consistency becomes a key requirement for ensuring correctness in the system [46].

In this section, we will first define the problem in maintaining cache consistency within such a disaggregated system. We will then present our cache consistency protocol, designed to address these challenges, and demonstrate its correctness to ensure that the cached data remains consistent and up-to-date with the storage layer.

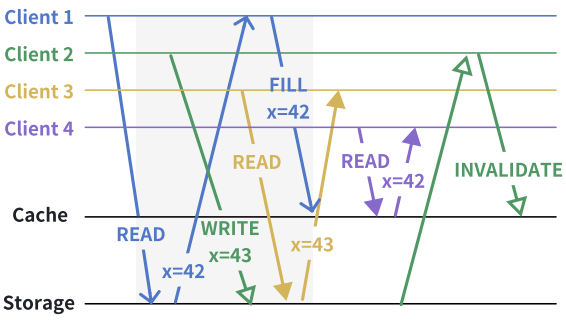
5.1 Problem Definition

To illustrate the problem with cache consistency, consider the time series diagrams shown in Figure 8, which illustrate how naive cache consistency protocols can violate the linearization guarantee.

Initially, the SSD holds the data $x = 42$, while the cache does not contain any record of x. Client 1 issues a read request for x, and since there is no cache hit, it fetches the data from the SSD and stores it in the cache. Meanwhile, Client 2 issues a write operation, updating the value of x to 43. Regarding write operations, two typical cache consistency protocols ex-



(a) The write-to-cache-first protocol violates linearizability.



(b) The write-to-SSD-first protocol violates linearizability.

Figure 8: The time series diagrams that demonstrate the naive cache protocols' violating the linearization guarantee.

ist, depending on the sequence of updating cache and the SSD storage: (1) writing to cache first and then updating the SSD; (2) writing to SSD first and then invalidating the cache item.

(1) The write-to-cache-first protocol may violate the linearizability guarantee, as illustrated in Figure 8 (a). Under this protocol, Client 2 will first update $x = 43$ to the cache. However, Client 1's later filling of $x = 42$ will overwrite the update, causing data loss.

Worse, suppose another Client 3 issues an eviction of x during the time difference, Client 1 will have no clue of Client 2's updating the cache item, remain unaware of the conflict, thus leading to data loss. As a consequence, when Client 4 finally reads x, it will get the previous value 42 rather than the latest value 43. The write-to-cache-first protocol cannot hold the linearizability guarantee.

(2) The write-to-SSD-first protocol also risks inconsistency in the event of cache refilling, as illustrated in Figure 8 (b). Under this protocol, Client 2 will first apply the update $x = 43$ to the SSD and later invalidate x from the cache. However, Client 3 performs a read operation for x after Client 2 updating the SSD but before Client 1 refilling the cache. At this point, Client 3 first accesses the cache, but since x is not present, it queries the SSD and retrieves the value $x = 43$ that was written by Client 2. This is then returned as the query result.

Then, suppose another Client 4 issues a read request for x after Client 1 has completed its operations. It reads from the cache and receives the value $x = 42$, which was cached earlier by Client 1. As we can see, this outcome violates the linearizability guarantee, as different clients observing the same key x at different times see inconsistent values.

The problem lies in the window of time during which the cached data has not been updated with the new value, while the SSD has already been updated with the new data. During this period, any client that attempts to query the cache may end up in an inconsistent state, leading to conflicting results. Specifically, the cache may not reflect the most recent updates on the SSD, causing clients to read stale data from the cache or the SSD, depending on the timing of their requests.

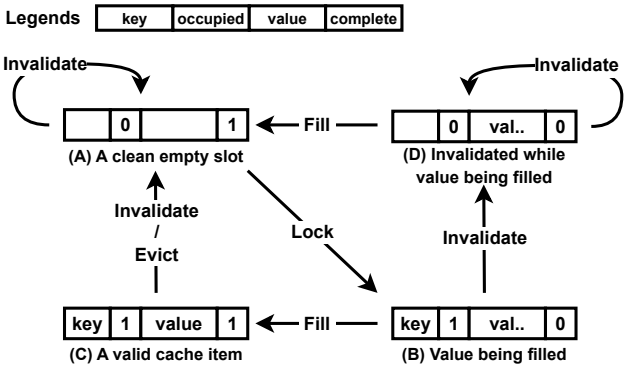


Figure 9: The states and transitions of Scalio's cache consistency protocol.

5.2 Protocol Design

To address the cache consistency issue, we propose a cache consistency protocol for Scalio. The core of this protocol is based on two flags — occupied and complete — which are added to each slot in the in-memory data structure, as shown in Figure 7 (b). The occupied flag indicates whether the slot contains a valid key, while the complete flag indicates whether the value in the slot is fully written.

These two flags result in four distinct states, as illustrated in Figure 9. Each state represents a specific condition of the cache slot:

State A: A slot is empty and safe to be reused when occupied is false and complete is true. In this state, the slot is not holding any valid data and can be safely reclaimed by other operations for filling new data.

State B: A slot is in the process of being filled. This indicates that the data in the cache is not yet fully valid and may lag behind the data in the SSD storage. If another client tries to access the same key while the slot is in State B, it must wait and retry, as the cache is not yet in a consistent state.

State C: A slot is valid and complete when both flags are true.

This state indicates that the key-value pair is fully written and consistent with the data in the SSD.

State D: A slot enters State D when the cache entry is invalidated by an update while the slot is still in State B. In this state, both flags are set to false, signifying that the slot is in an inconsistent state. Any other reader that attempts to query the item will treat the item as a miss, and any other write will be blocked until the client issuing the cache filling has completed writing the value (transiting to State A), thus avoiding potential collisions or torn reads/writes.

This state-based design is the foundation of our cache consistency protocol, ensuring that the data in the cache remains consistent with the SSD storage. The protocol's operation is integrated into the system workflow, and the detailed sequence of steps is outlined in the following paragraphs.

Read workflow. When a client initiates a read request, it starts by performing an RDMA read of the hash block (Step 1 in Figure 7 (b)). The client first searches for a slot containing the matching key. If the key is found in State C, the corresponding item is returned. If the key is in State B, the client will wait and retry later, as the cached item is not yet valid. If no match is found, a cache miss occurs, and the client proceeds with the following steps.

The client continues by selecting a victim slot according to the client-centric LRU eviction policy (§ 4.1). It issues an RDMA CAS operation to update the complete flag to 0 (Step 2 in Figure 7 (b)) and attempts to lock the slot. If the CAS operation succeeds, the client writes the key to the slot and sets the occupied flag to 1, transitioning the slot to State B.

Since clients perform cache-related operations independently, chances are that two clients may happen to choose two different slots as victim slots. To ensure atomicity and avoid key collisions, the client performs a double-read (Step 3 in Figure 7 (b)). A second RDMA read checks if any other slot in the hash block contains the same key and has the occupied flag set to 1. If a collision is detected, the client resets the slot's flags (occupied = 0, complete = 1) and reverts to Step 1, ensuring only one client is filling with the same key. If no collision occurs, the lock process is successfully completed, and the client proceeds to retrieve the offset of the SSD index from the block, moving on to Step 4.

After completing Step 4 and obtaining the data from the SSD, the client fills the value into the selected slot and updates the complete flag to 1 via RDMA write, transitioning the slot to State C.

To handle client failures, Scalio can opt in a lease mechanism that bounds how long a client may hold or update a cache slot. Each key-value slot maintains an additional timeout_timestamp alongside its state flags, representing the latest time the client's lease remains valid. When a client acquires a slot (entering State B), it writes the current timestamp plus a lease duration into timeout_timestamp. If that timestamp is exceeded—indicating the client has failed—other clients will see the expired lease, treat the slot as unlocked, and automatically evict it for reuse. This approach ensures that orphaned locks do not persist indefinitely and that resources are safely reclaimed without impacting Scalio's consistency guarantees.

Write workflow. For write requests, once the client receives confirmation of the successful write from the server (Step 3 in Figure 7 (c)), it invalidates any outdated cache entries. The client performs an RDMA read on the corresponding hash block and checks if any slot contains the same key with the occupied flag set to 1 (either State C or State B). The client then sets the occupied flag to 0, transitioning the slot to State A or State D, as appropriate.

Under our cache consistency protocol, the consistency problem is resolved as follows: Client 1, after reading the cache layer, encounters a cache miss and proceeds to lock a slot. We now focus on the gray-shaded section illustrated in Figure 8 (b). Since Client 1 has locked a slot, the slot will be in State B within this section. At this point, according to the protocol, Client 3 will attempt a read but will wait briefly and then retry. Eventually, the cache will have been updated by Client 1, turning the slot to state C, and it will read the value $x = 42$ written by Client 1, which will be returned as the result of the query. We can then linearly order the clients as Client 1, 3, 4, 2 and ensure that the operations behave in a way that respects the order in which the clients perform their actions.

5.3 Correctness

We now demonstrate the correctness of Scalio's cache consistency protocol, by two steps: (1) define the linearization points of the read and write operations; (2) show that such definition can linearize all operations.

We begin with the definition of the linearization points. The linearization point of a read operation is defined as: (1) on cache hit, the moment when the RDMA read reaches the cache and gets the hash block (i.e., when Step 1 is done in Figure 7 (b)), or (2) on cache miss, when the victim slot has been locked and successfully validated by the double-read check (i.e., when Step 3 is done in Figure 7 (b)). The linearization point of a write operation is defined as the earliest moment when no cache slot in State B or State C exists with the same key after the update is applied to the SSD. It is easy to verify that by this definition, the linearization points are between the the start and the end of each operation.

In order to prove linearizability, the key is to demonstrate that all operations appear to behave in the sequence specified by the definition. Since the operations are all key-value read and write operations, it is equivalent to showing that every read operation is guaranteed to read **exactly** the value updated by the closest write before, as illustrated in Figure 10.

Part 1: every read operation will never read values earlier than its closest-write-before's update. At the linearization point of the closest-write-before, the corresponding cache slot should be in either State A or State D, by the definition of the linearization point. If the read operation is a cache hit,

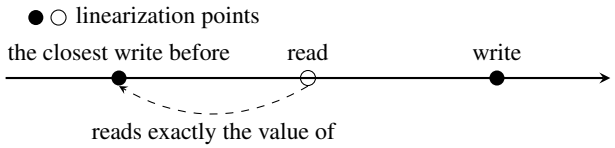


Figure 10: Illustration of the proof objective.

the cache slot must have been transitted to State C at the time of the cache hit. The only possible path of the transition is by another client that locks the slot, and refills the slot with the SSD data at the time of locking. The whole process of transition happens after the writer's linearization point, so the data read by the cache hit must be no earlier than the data written by the closest-write-before. On the other hand, if the read operation is a cache miss, by definition, it will read the closest-write-before's already applied update on the SSD.

Part 2: every read operation will never read values later than its closest-write-before's update. This is guaranteed by the definition of the write operations' linearization point, because at any moment before a subsequent write operation's linearization point, the subsequent update is either not on the SSD or masked by an existing cache slot with the same key but a previous value. Therefore, the read operation will never read newer values that have not been linearized yet.

By combining the two parts, it can be assured that all read and write operations happen in the sequence by the definition of their linearization point. Since each linearization point happens between the corresponding operation's invoke and return, it can be further concluded that if an operation ends before another operation begins, the linearization point of the latter operation must be later than the other. In other words, the linearization point definition respects real time ordering.

In conclusion, the linearization property of Scalio's cache consistency protocol is demonstrated.

6 Evaluation

6.1 Experimental Methodology

Testbed. We configured our testbed to resemble a DPU-based JBOF storage environment, consisting of one storage node and five client nodes (with up to 160 client threads) interconnected via a high-speed RDMA network. The storage node includes seven Samsung 970 PRO SSDs [49], each offering 500K IOPS under 4KB blocks, connected to a ConnectX-6 HCA. We emulated a DPU-based JBOF configuration by limiting the available Intel Xeon Gold CPU cores to 8 and memory to 8 GB, which aligns with the configuration of commodity DPU-based JBOF platforms [6, 16, 35, 39].

Implementation details. Our system environment is based on Linux kernel version 5.15.0, which includes native support for NVMe-oF targets. We installed the MLNX_OFED driver (version 24.04-0.7.0) and set up the NVMe target subsystem (nvmet) to handle NVMe-oF requests, configuring it to use RDMA as the transport protocol. Additionally, we enabled the `attr_offload` parameter within the NVMe target setup to support the offloading implementation for NVMe-oF, which allows for scalable handling of I/O operations by bypassing traditional CPU processing [40, 41]. For the software stack, we further maintained fairness by porting LEED's code using the same SPDK stack and employed the same SSD storage data structures as LEED. The source code is available at https://github.com/madsys-dev/scalio-osdi25-ae.

Workloads. For workload evaluation, we employed the Yahoo! Cloud Serving Benchmark (YCSB) [11], focusing on five specific workload types: YCSB A, YCSB B, YCSB C, YCSB D and YCSB F. YCSB E is omitted as other systems because it requires range query. We generated 20 million key-value pairs, each with a maximum key length of 16 bytes and a maximum value length of 64 bytes, distributed with a Zipfian distribution and a Zipfian constant of 0.99. The cache can hold approximately 2 million key-value pairs.

Baselines. We compared Scalio with two setups: (1) the LEED [16] JBOF key-value storage system (denoted as LEED) and (2) the LEED storage system combined with the Ditto [52] disaggregated key-value caching system (denoted as LEED+Ditto). For fairness, we apply the same Arm-core-allocation strategy as LEED, accompanied with the round-robin SSD binding strategy as illustrated in Figure 4. The cache size is set to 1 GB for both Scalio and the LEED+Ditto baseline.

6.2 Performance Improvements

We evaluated the performance of Scalio along with the baseline systems by varying the number of SSDs from 1 to 7 and measuring throughput under YCSB workloads A, B, C, D, and F. The results, shown in Figure 11, highlight Scalio's superior throughput and scalability across all workloads as the number of SSDs increases.

Scalio demonstrates superior scalability compared to the baselines, as the baseline systems experience throughput saturation when the number of SSDs reaches 4, while continues to scale efficiently as more SSDs are added. Under read-intensive and read-only workloads (YCSB B, C, D), Scalio achieves up to 3× speedup, and under write-intensive workloads (YCSB A, F), it achieves up to 2× speedup.

The 3× speedup observed in read workloads is primarily due to the use of NVMe-oF target offload. This optimization frees the JBOF server's CPU from handling read operations, allowing it to fully utilize the SSDs' I/O capabilities, as shown in Figure 12. In contrast, the baseline systems struggle to fully utilize SSD IOPS because their JBOF servers become CPU-bound as the number of SSDs increases.

For write workloads, the 2× speedup is largely attributed to the batched write mechanism. By combining multiple value updates into a single SSD write operation, Scalio reduces CPU

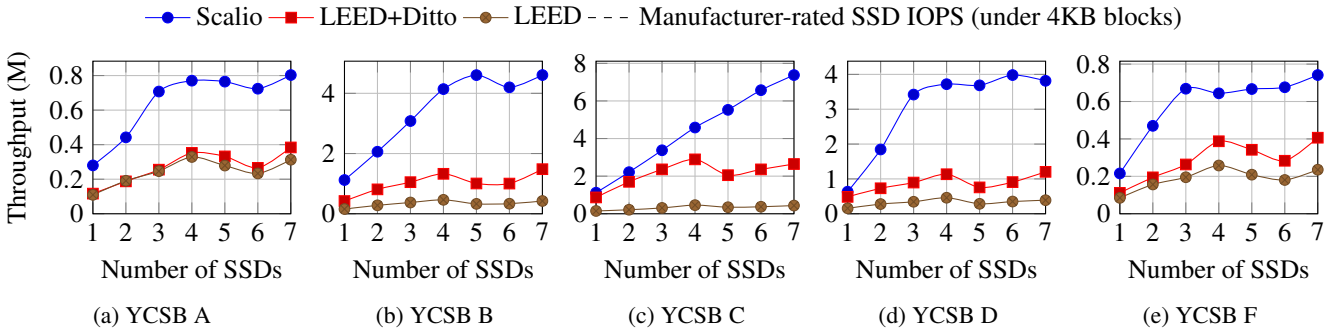


Figure 11: Throughput for YCSB workloads A,B,C,D and F as the number of SSDs scale up.

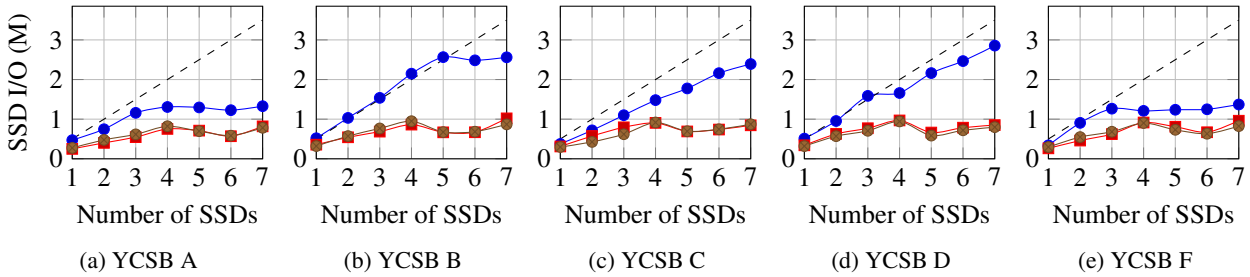


Figure 12: SSD I/O usage under YCSB workloads A,B,C,D and F as the number of SSDs scale up.

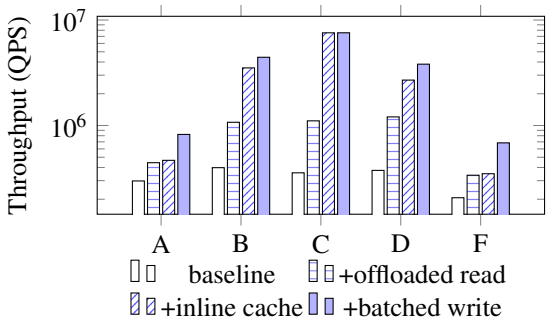


Figure 13: Throughput improvement breakdown under YCSB workloads A, B, C, D, and F. The Y-axis is log scale.

overhead, which improves SSD I/O efficiency. The batched write mechanism minimizes the number of write operations required, thus alleviating pressure on the JBOF server and enhancing throughput.

A more detailed breakdown of these improvements can be found in § 6.3, with performance impacts of each optimization evaluated and discussed.

In conclusion, Scalio's optimizations—primarily inline cache, offloaded reads and batched writes—effectively reduce CPU overhead, maximize SSD IOPS utilization, and enhance scalability and throughput as the number of SSDs increases. These improvements enable Scalio to outperform baseline systems in both throughput and resource utilization.

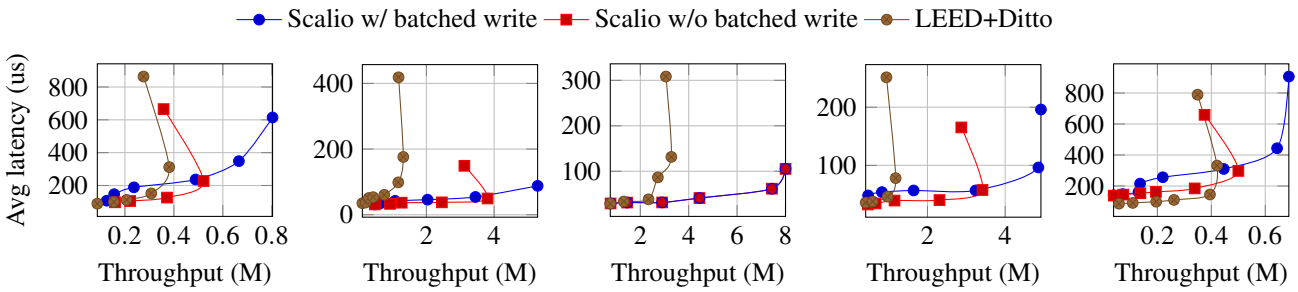
6.3 Improvements Breakdown

To assess the impact of each individual optimization, we conducted a detailed analysis of Scalio's contributions under YCSB workloads A, B, C, D, and F, using a 7-SSD configuration. Figure 13 shows the breakdown of performance improvements as we incrementally enabled each optimization: (1) offloaded read, (2) inline cache, and (3) batched write. By evaluating the effects of these optimizations, we demonstrate how each one contributes to the overall system performance.

Offloaded read. Offloaded read provides a speedup of 1.5× to 3.2× across all workloads by offloading massive SSD I/O to the network, relieving the DPU's CPU. This increases SSD I/O utilization and bridges the gap between the SSD's I/O capabilities and the system's performance, as in Figure 12.

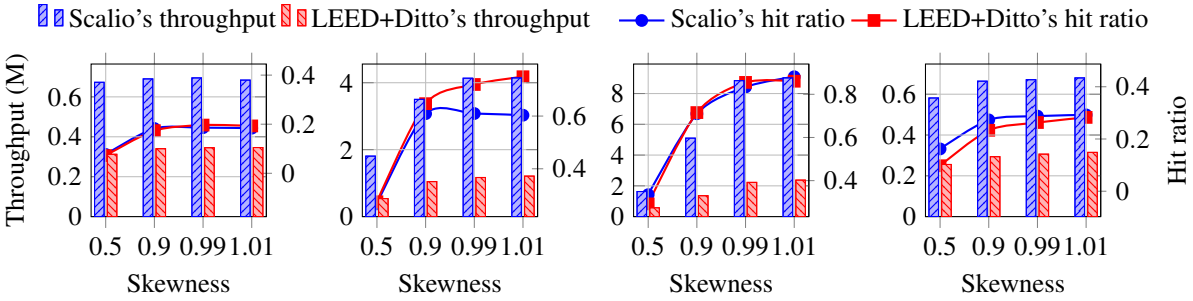
Inline cache. Inline caching is particularly effective under read-only and read-intensive workloads by absorbing hot reads. Our system statistics show that Scalio's in-memory caching layer absorbs 72.2%, 85.2%, and 62.6% of total read and write operations under YCSB B, C, and D, respectively. This absorption leads to throughput improvements of 3.6×, 6.7× and 2.7×, matching the observed speedup.

Batched write. Batched write provides a speedup of up to 1.96×, and its contribution is the most obvious in YCSB A and F, which belong to write-intensive workloads. According to our experiments, batched write can reduce the number of SSD writes per key-value update operation from 2 to approximately 1, which reduces the burden on the DPU's CPU, thus contributing to the speedup.



(a) YCSB A (b) YCSB B (c) YCSB C (d) YCSB D (e) YCSB F

Figure 14: Latency to throughput by varying the number of concurrent I/O queues.



(a) YCSB A (b) YCSB B (c) YCSB C (d) YCSB F

Figure 15: Throughput and cache hit ratio when varying skewness under YCSB A, B, C, and F.

6.4 Latency

In this subsection, we compare the latency performance of Scalio with and without the batched write feature, alongside the LEED+Ditto baseline. The comparison highlights the trade-off between latency and throughput introduced by the batched write mechanism, where throughput is significantly improved at an acceptable cost of some latency. Our evaluation demonstrates two key findings: (1) Scalio without batched write delivers both lower latency and higher throughput compared to LEED+Ditto, and (2) Scalio with batched write achieves even higher throughput, with an acceptable latency trade-off. Users can choose between prioritizing low latency in latency-critical scenarios or enabling batched write for higher throughput.

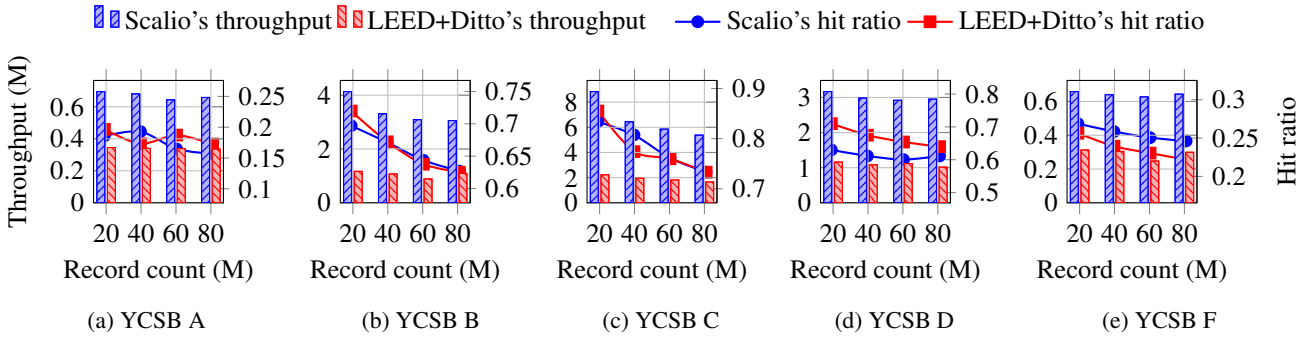
Comparing Scalio without the batched write mechanism to the LEED+Ditto baseline, Scalio demonstrates a 20% to 30% reduction in average latencies across all workloads, while also achieving a 2× to 3× throughput improvement. Both the reduction in latency and the throughput improvement are due to two key optimizations: Scalio's inline caching, which reduces round-trip time on cache hits by using a compact inline data structure, and the bypassing of the JBOF server's CPU, which lowers network latency on cache miss paths. These optimizations not only alleviate the DPU's CPU load but also improve SSD I/O utilization, driving higher throughput with lower latency and enhancing system performance. Under the read-modify-write workload YCSB F, Scalio ensures cache consistency, which the baseline system fails to achieve, at slight expense of latency.

Comparing Scalio with the batched write mechanism to the LEED+Ditto baseline, we observe a trade-off where the group commit mechanism introduces significant throughput improvements in exchange for a tolerable increase in latency. For instance, in YCSB A, Scalio achieves 2.1× higher throughput (800 M IOPS) compared to the baseline, with only a 1.97× increase in latency (614 us). This shows that while there is a moderate latency penalty, it is well-compensated by the substantial gain in throughput, making the trade-off beneficial for overall system performance.

In conclusion, Scalio provides a well-balanced approach to system performance, effectively addressing the trade-offs between throughput and latency. When batched write is disabled, Scalio outperforms the baseline in both throughput and latency, providing lower latencies across all workloads. With batched write enabled, Scalio achieves much higher throughput with only a minimal increase in latency, allowing it to efficiently handle diverse workloads while maintaining an optimal balance between performance and responsiveness.

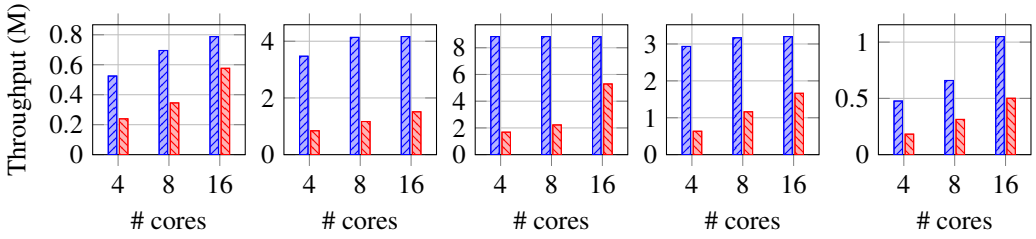
6.5 Sensitivity Analysis

In this subsection, we conduct a sensitivity analysis under the 7-SSD configuration by varying several key parameters. We



(a) YCSB A (b) YCSB B (c) YCSB C (d) YCSB D (e) YCSB F

Figure 16: Throughput and cache hit ratio when varying the total number of records under YCSB A, B, C, D, and F.



(a) YCSB A (b) YCSB B (c) YCSB C (d) YCSB D (e) YCSB F

Figure 17: Throughput when varying the number of cores under YCSB A, B, C, D, and F.

will present the results of four experiments: (1) varying the workloads' skewness , (2) varying the dataset size, (3) varying the number of CPU cores, and (4) varying the hash block size.

Skewness. We varied the Zipfian constant from 0.5 to 1.01 to evaluate the system performance under different workload skewnesses, as shown in Figure 15. (YCSB D is omitted as it does not follow the Zipfian distribution.) As observed, Scalio performs better as the skewness increases, particularly in read-only and read-intensive workloads, where higher skewness results in a higher hit ratio. For write-intensive workloads, while hot data may hinder performance, this affects both Scalio and the baseline similarly, with Scalio still outperforming the baseline. Overall, Scalio consistently outperforms the baseline system of 2× to 3× across all skewness levels and workloads, demonstrating its ability to handle varying workload distributions effectively.

Dataset size. We varied the total number of key-value pairs from 20 million to 80 million, testing the system's performance at each step, as shown in Figure 16. As the dataset size increases, the cache hit ratio slightly decreases and throughput marginally drops. However, Scalio consistently outperforms the baseline, demonstrating its robustness with larger datasets.

Number of CPU cores. We varied the number of CPU cores, testing configurations with 4, 8, and 16 cores. The 16-core configuration represents an ideal scenario, assessing the system's performance under ample CPU resources. The results are shown in Figure 17. Scalio is less sensitive to variations in the number of CPU cores, especially in read-only or read-intensive workloads, as it offloads operations to the network and hence requires fewer CPU resources. In contrast, the performance of LEED+Ditto degrades significantly as the number of CPU cores decreases. The performance gap between Scalio and the baseline widens from 1× to 2× (under the 16-core configuration) to 2× to 5× (under the 4-core configuration), highlighting Scalio's advantage in environments with limited CPU resources.

Capacity of hash block. We varied the hash block's capacity by adjusting the number of slots per block, testing configurations with 4, 8, and 16 slots. As shown in Figure 18, changes in block size have minimal impact on Scalio's performance. This suggests that the hash block size can be tuned according to specific cache requirements, offering flexibility in optimizing performance without significant trade-offs.

7 Discussion

Fairness of evaluation setup. While our evaluation uses an emulated DPU setup, it provides a conservative and fair estimate of real-world performance. We constrained the testbed to 8 cores and 8 GB memory to match typical DPU resource limits, and Scalio's core performance benefits stem from offloading CPU-intensive SSD I/O via NVMe-oF target offload—an advantage that holds regardless of the CPU power. We further ensured fairness by aligning our setup with LEED's, using the same SPDK stack, RDMA topology, and data structures. This setup realistically reflects Scalio's deployment scenario

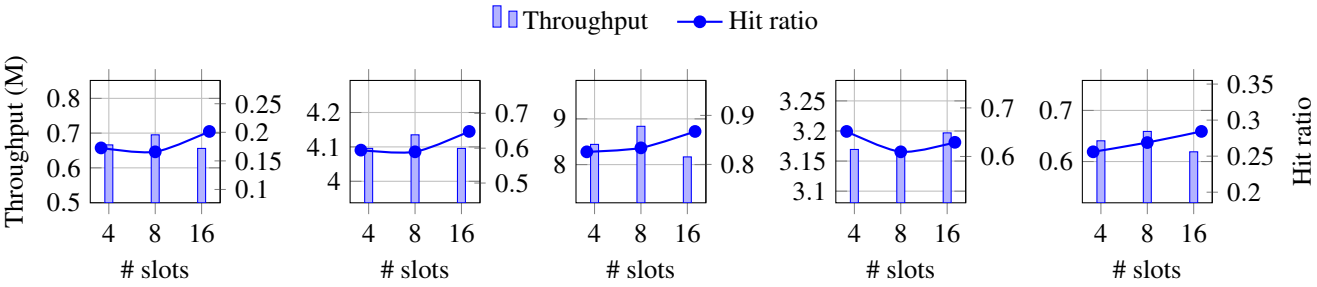


Figure 18: Throughput and cache hit ratio when varying the number of slots per hash block under YCSB A, B, C, D, and F.

and expected performance characteristics.

Energy efficiency of DPU-based design. Scalio's DPU-based approach leverages ASIC-accelerated NVMe I/O to achieve significantly higher energy efficiency than general-purpose CPUs. While increasing CPU power consumption might suggest potential performance gains, it does not guarantee linear scaling—especially for I/O-heavy workloads where overheads grow disproportionately. In contrast, DPU-based JBOF systems can already efficiently support 26 or more SSDs with lower power. Given these factors, Scalio's DPU-based design offers higher effective performance per watt and better scalability than simply scaling up CPU resources.

Server failure handling. Scalio ensures robustness against server-side failures by integrating with orthogonal redundancy techniques such as RAID or dual-DPU setups at the JBOF level. These mechanisms are widely adopted in disaggregated storage systems and can tolerate SSD or node failures without modifying Scalio's core protocol. For example, RAID-10 can preserve data availability and consistency even under partial disk failures, allowing Scalio to continue operating seamlessly. This integration enhances system reliability while preserving Scalio's scalability and performance benefits.

8 Related Work

Energy-efficient storage. Energy-efficient storage solutions are increasingly necessary as storage demands rise. LEED [16] uses DPU-based JBOF platforms for lower power consumption, but its performance is limited by the DPU's CPU load. FAWN [2] relies on low-power "wimpy" nodes, but its design is not well-suited for modern high-density JBOF architectures, which offer better energy efficiency. Gimbal [35] leverages DPU-based JBOFs for managing tenants with strong QoS guarantees, but its focus is on tenant isolation, not computational efficiency. TrickleKV [62], a key-value storage system built on JBOF, aims to alleviate network traffic contention rather than improving CPU efficiency. SKV [55] offloads replication to SmartNICs in disaggregated key-value systems but still relies on host servers for most operations. Scalio overcomes the limitations of these systems by increasing storage density and offloading tasks to reduce CPU load on the DPU, significantly enhancing overall power efficiency.

Storage over network. Storage over network technologies enable the separation of storage and computation by connecting devices via protocols such as iSCSI and NVMe-over-Fabrics (NVMe-oF). iSCSI [26] facilitates storage networking through TCP/IP, but it suffers from CPU inefficiencies due to single-queue structures, creating bottlenecks, particularly with multi-core CPUs. NVMe-oF improves upon iSCSI by utilizing protocols like RDMA for direct data transfer between SSDs and compute nodes, as explored in [17]. However, the standard NVMe-oF implementation still places a heavy load on target CPUs which limits scalability, and there has been few studies on the NVMe-oF target offload implementation.

Cache consistency/coherence protocols. Cache consistency/coherence protocols are well-studied in both monolithic and distributed systems. In monolithic systems, protocols like MESI [20] ensure consistency between L1/L2 caches and main memory. In distributed systems, various protocols ensure consistency between cached data and underlying storage. DistCache [29] partitions hot objects across cache nodes using independent hash functions and routes queries with a power-of-two-choices mechanism. Similarly, CliqueMap [54] uses a quorum-based approach, combining replication, client-based quoruming, and self-validating server responses. These protocols maintain consistency across distributed nodes. Scalio's protocol, on the other hand, is designed for DPU-based disaggregated systems, where RDMA reduces CPU overhead and ensures consistency through efficient cache management without full synchronization across all nodes.

Disaggregated in-memory key-value stores. High-performance, disaggregated in-memory key-value stores typically use one-sided RDMA to bypass the CPU, achieving efficient, direct data access. FORD [64] provides transaction support using RDMA atomic operations, but this design is more lock-heavy and less efficient in cases without transactional needs, such as in-memory index. Ditto [52] offers a more efficient solution in terms of in-memory caching, but it lacks integration with secondary storage layers like the SSD storage layer, and thus cannot be directly applied to systems

with both DRAM and SSD storage. Our work targets efficient serving of small key-value pairs, along with integration with SSD storage layers to provide high-performance key value storage service with linearizability guarantee.

9 Conclusion

In this paper, we propose Scalio, a scalable disaggregated key-value store designed for high-density JBOF systems. By leveraging NVMe-oF Target Offload, Scalio bypasses the DPU's CPU, reducing overhead, while its two-layer architecture and batched write mechanism efficiently handle bursty workloads. Our cache consistency protocol ensures data consistency across the system. Experimental results demonstrate Scalio's superior scalability and performance, particularly as the number of SSDs increases.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mr. Sanidhya Kashyap, for their valuable feedback. The authors affiliated with Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (92467102) and Tsinghua University Initiative Scientific Research Program, Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), and Shandong Provincial Natural Science Foundation (ZR2023LZH014).

References

[1] AIC. AIC Inc. to Showcase Advanced Server and Storage Solutions at SC24. https://www.aicipc.com/en/news/6107, 2024.

[2] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 1–14.

[3] Antonopoulos, P., Budovski, A., Diaconu, C., Hernandez Saenz, A., Hu, J., Kodavalla, H., Kossmann, D., Lingam, S., Minhas, U. F., Prakash, N., Purohit, V., Qu, H., Ravella, C. S., Reisteter, K., Shrotri, S., Tang, D., and Wakade, V. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1743–1756.

[4] Barsellotti, L., Alhamed, F., Olmos, J. J. V., Paolucci, F., Castoldi, P., and Cugini, F. Introducing data processing units (DPU) at the edge. In *2022 International Conference on Computer Communications and Networks (ICCCN)* (2022), IEEE, pp. 1–6.

[5] Broadcom. Broadcom Announces Production Availability of Industry's First 100G Programmable Storage Adapter with NVMe-oF Support. https://www.broadcom.com/company/news/product-releases/40966, 2019.

[6] Broadcom. NVMe over Fabrics Performance. https://docs.broadcom.com/doc/broadcom-stingray-100G-NVMe-oF-performance, 2019.

[7] Burke, M., Dharanipragada, S., Joyner, S., Szekeres, A., Nelson, J., Zhang, I., and Ports, D. R. PRISM: Rethinking the RDMA interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 228–242.

[8] Cao, Q., Lyu, W., Wang, X. R., Guan, X., Wang, L., Yan, S., Wu, T., and Wang, X. Nonvolatile Multistates Memories for High-density Data Storage. *ACS Applied Materials & Interfaces 12*, 38 (2020), 42449–42471.

[9] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 275–290.

[10] Chen, Y., Wei, X., Shi, J., Chen, R., and Chen, H. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), pp. 1–17.

[11] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[12] Couvert, P., and Marketing, D. P. High Speed I/O Processor for NVMe over Fabric. *Flash Memory Summit* (2016).

[13] Di Girolamo, S., De Sensi, D., Taranov, K., Malesevic, M., Besta, M., Schneider, T., Kistler, S., and Hoefler, T. Building blocks for network-accelerated distributed file systems. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (2022), IEEE, pp. 1–14.

[14] Express, N. Accelerating NVMe™ over Fabrics with Hardware Offloads at 100Gb/s and Beyond. https://nvmexpress.org/wp-content/uploads/Accelerating-NVMe-over-Fabrics-with-Hardware-Offloads.pdf, 2017.

[15] Guo, Z., Lin, J., Bai, Y., Kim, D., Swift, M., Akella, A., and Liu, M. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (2023), pp. 916–929.

[16] Guo, Z., Zhang, H., Zhao, C., Bai, Y., Swift, M., and Liu, M. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference* (2023), pp. 1012–1027.

[17] Guz, Z., Li, H., Shayesteh, A., and Balakrishnan, V. Performance characterization of NVMe-over-Fabrics storage disaggregation. *ACM Transactions on Storage (TOS) 14*, 4 (2018), 1–18.

[18] Intel. Intel Optane DC SSD Series. https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dc-ssd-series.html, 2024.

[19] Intel. Intel® Xeon® Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon.html, 2024.

[20] Ivanov, L., and Nunna, R. Modeling and verification of cache coherence protocols. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)* (2001), vol. 5, pp. 129–132 vol. 5.

[21] Jin, Y. T., Ahn, S., and Lee, S. Performance Analysis of NVMe SSD-based All-flash Array Systems. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2018), IEEE, pp. 12–21.

[22] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), pp. 437–450.

[23] KFOURY, E. F., CHOUEIRI, S., MAZLOUM, A., ALSABEH, A., GOMEZ, J., AND CRICHIGNO, J. A Comprehensive Survey on SmartNICs: Architectures, Development Models, Applications, and Research Directions. *IEEE Access 12* (2024), 107297–107336.

[24] KFOURY, E. F., CHOUEIRI, S., MAZLOUM, A., ALSABEH, A., GOMEZ, J., AND CRICHIGNO, J. A comprehensive survey on SmartNICs: Architectures, development models, applications, and research directions. *IEEE Access* (2024).

[25] KIM, B., KIM, J., AND NOH, S. H. Managing Array of SSDs When the Storage Device Is No Longer the Performance Bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (2017).

[26] KLIMOVIC, A., KOZYRAKIS, C., THERESKA, E., JOHN, B., AND KUMAR, S. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), pp. 1–15.

[27] LI, C., CHEN, H., RUAN, C., MA, X., AND XU, Y. Leveraging NVMe SSDs for building a fast, cost-effective, LSM-tree-based KV store. *ACM Transactions on Storage (TOS) 17*, 4 (2021), 1–29.

[28] LISS, L. NVMf target offload. https://www.openfabrics.org/images/2018workshop/presentations/308_LLiss_NVMfTargetOffload.pdf, 2018.

[29] LIU, Z., BAI, Z., LIU, Z., LI, X., KIM, C., BRAVERMAN, V., JIN, X., AND STOICA, I. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019), pp. 143–157.

[30] LUO, J., LIU, H., HE, Y., VARGAS-ROSALES, C., AND FAN, L. High-Density NVMe SSD With DRAM-Less eRAID Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023).

[31] MA, S., MA, T., CHEN, K., AND WU, Y. A Survey of Storage Systems in the RDMA era. *IEEE Transactions on Parallel and Distributed Systems 33*, 12 (2022), 4395–4409.

[32] MCALLISTER, S., BERG, B., TUTUNCU-MACIAS, J., YANG, J., GUNASEKAR, S., LU, J., BERGER, D. S., BECKMANN, N., AND GANGER, G. R. Kangaroo: Theory and Practice of Caching Billions of Tiny Objects on Flash. *ACM Trans. Storage 18*, 3 (Aug. 2022).

[33] MICRO, S. Petascale JBOF All-Flash Array for AI Data Pipeline Acceleration. https://www.supermicro.org.cn/en/products/jbof, 2024.

[34] MICRON. Ethernet bunch of flash in an NVMe-oF™ network for low-cost storage at scale. https://www.micron.com/about/blog/storage/ssd/ethernet-bunch-of-flash-in-nvme-of-network, 2020.

[35] MIN, J., LIU, M., CHUGH, T., ZHAO, C., WEI, A., DOH, I. H., AND KRISHNAMURTHY, A. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 106–122.

[36] MYUNG, K., KIM, S., YEOM, H. Y., AND PARK, J. Efficient and scalable external sort framework for NVMe SSD. *IEEE Transactions on Computers 70*, 12 (2020), 2211–2217.

[37] NGUYEN, T. A., JEON, H., HAN, D., BAE, D.-H., YU, Y. J., KIM, K., PARK, S., JEONG, J., AND NAM, B. NVMe-driven lazy cache coherence for immutable data with NVMe over Fabrics. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)* (2023), IEEE, pp. 394–400.

[38] NVIDIA. BF1600 NVIDIA® Mellanox® BlueField® Dual Port 100Gb/s Controller Card for InfiniBand and Ethernet. https://network.nvidia.com/files/doc-2020/pb-bluefield-storage-controller-card-vpi.pdf, 2020.

[39] NVIDIA. NVIDIA BlueField Networking Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2021.

[40] NVIDIA. HowTo Configure NVMe over Fabrics (NVMe-oF) Target Offload. https://enterprise-support.nvidia.com/s/article/howto-configure-nvme-over-fabrics--nvme-of--target-offload, 2022.

[41] NVIDIA. Simple NVMe-oF Target Offload Benchmark. https://enterprise-support.nvidia.com/s/article/simple-nvme-of-target-offload-benchmark, 2022.

[42] NVIDIA. NVIDIA BlueField-2 BF2500 Ethernet DPU Controller. https://docs.nvidia.com/networking/display/bf2endpucontroller/introduction, 2023.

[43] NVIDIA. Chelsio T7 DPU Furthers Expansive Ethernet Storage Networking, Empowering Open, ROI-Enhanced Enterprise Storage Platforms. https://www.chelsio.com/wp-content/uploads/resources/pr-chelsio-t7-jbof.pdf, 2024.

[44] NVIDIA. NVIDIA BlueField-3 Networking Platform User Guide. https://docs.nvidia.com/networking/display/bf3dpu/specifications, 2024.

[45] NVIDIA. NVIDIA ConnectX InfiniBand Adapters. https://www.nvidia.com/en-us/networking/infiniband-adapters/, 2024.

[46] PAN, L. Cache made consistent. https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/, 2022.

[47] PARK, S., KIM, Y., URGAONKAR, B., LEE, J., AND SEO, E. A comprehensive study of energy efficiency and performance of flash-based SSD. *Journal of Systems Architecture 57*, 4 (2011), 354–365.

[48] PI, R. Raspberry Pi 4 Model B. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/, 2021.

[49] SAMSUNG. Samsung SSD 970 PRO NVMe® M.2. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-pro-nvme-m2-1tb-mz-v7p1t0bw/, 2017.

[50] SAMSUNG. Samsung 983 DCT. https://semiconductor.samsung.com/consumer-storage/datacenter-ssd/983dct/, 2018.

[51] SCALEFLUX. AIC and ScaleFlux Unveil New Storage Array Based on NVIDIA BlueField-3 DPU. https://scaleflux.com/in-the-media/aic-and-scaleflux-unveil-new-storage-array-based-on-nvidia-bluefield-3-dpu/, 2024.

[52] SHEN, J., ZUO, P., LUO, X., SU, Y., GU, J., FENG, H., ZHOU, Y., AND LYU, M. R. Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles* (2023), pp. 675–691.

[53] SHIM, H. PHash: A memory-efficient, high-performance key-value store for large-scale data-intensive applications. *Journal of Systems and Software 123* (2017), 33–44.

[54] SINGHVI, A., AKELLA, A., ANDERSON, M., CAUBLE, R., DESHMUKH, H., GIBSON, D., MARTIN, M. M., STROMINGER, A., WENISCH, T. F., AND VAHDAT, A. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 93–105.

[55] SUN, S., ZHANG, R., YAN, M., AND WU, J. SKV: A SmartNIC-Offloaded Distributed Key-Value Store. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)* (2022), IEEE, pp. 1–11.

[56] SUN, S., ZHANG, R., YAN, M., AND WU, J. Offloading Distributed Key-Value Stores with Off-Path SmartNICs. *Available at SSRN 4877692* (2024).

[57] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 233–251.

[58] XU, J., QIU, Y., CHEN, Y., WANG, Y., LIN, W., LIN, Y., ZHAO, S., LIU, Y., WANG, Y., AND CHEN, W. Performance Characterization of SmartNIC NVMe-over-Fabrics Target Offloading. In *Proceedings of the 17th ACM International Systems and Storage Conference* (2024), pp. 14–24.

[59] YANG, J., YUE, Y., AND RASHMI, K. V. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 191–208.

[60] YANG, Z., HARRIS, J. R., WALKER, B., VERKAMP, D., LIU, C., CHANG, C., CAO, G., STERN, J., VERMA, V., AND PAUL, L. E. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2017), IEEE, pp. 154–161.

[61] YOON, D. Y., CHOWDHURY, M., AND MOZAFARI, B. Distributed lock management with RDMA: decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1571–1586.

[62] ZHAN, L., LU, K., XIONG, Y., WAN, J., AND YANG, Z. TrickleKV: A High-Performance Key-Value Store on Disaggregated Storage with Low Network Traffic. *IEEE Access* (2024).

[63] ZHANG, J., SHIHAB, M., AND JUNG, M. Power, Energy, and Thermal Considerations in SSD-Based I/O Acceleration. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, June 2014), USENIX Association.

[64] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (2022), pp. 51–68.



Low End-to-End Latency atop a Speculative Shared Log with Fix-Ante Ordering

Shreesha G. Bhat, Tony Hong, Xuhao Luo, Jiyu Hu, Aishwarya Ganesan, Ramnatthan Alagappan

University of Illinois Urbana-Champaign

Abstract. Today's shared logs incur expensive coordination to globally order records across storage shards before they can deliver records to applications. This makes them unsuitable for many modern applications that must process ingested data as early as possible and realize low *end-to-end* (e2e) latencies. We propose *SpecLog*, a new shared log abstraction that delivers records by speculating the global order, allowing the application's computation and shared-log coordination to be overlapped, thus reducing e2e latency. To enable accurate speculations, we introduce *fix-ante ordering*, a novel ordering mechanism that predetermines the global order and makes the shards adhere to the predetermined order. With fix-ante ordering, shards, except in rare cases, can accurately predict where their records will sit in the total order before global coordination. We build Belfast, an implementation of the SpecLog abstraction and fix-ante ordering. Our experiments show that Belfast offers lower e2e latencies than current shared logs while preserving their elasticity, flexibility, and scalability.

1 Introduction

Shared logs play a central role in many modern, data-driven applications like high-speed trading [27, 28], real-time search [54, 60], IoT analytics [24, 49], fraud detection [15, 46], and others [3, 59]. At a high level, in all these applications, a set of upstream components ingest data into the shared log, and a set of downstream tasks consume and process the data.

Shared logs ease the construction of the above applications by providing an abstraction of a durable and ordered sequence of records. Many shared logs [1, 4, 16, 29, 36, 61] provide this abstraction while storing records on multiple storage shards, i.e., they offer a *total order* of records across shards.

Early total-order logs like Corfu [16, 17] first obtain a position from a sequencer and then use a fixed position-to-shard mapping to write to the shard responsible for that position. This architecture fundamentally leads to three problems for applications [29, §2.2]: inability to seamlessly add or remove shards (to elastically adapt to demands), inflexible data placement, and limited scalability. Fortunately, a family of shared logs [29, 32, 36], starting from Scalog [29], address these problems by taking a *durability-first* approach. Here, clients first make records durable on the shards of their choice; the shards then periodically contact a sequencing layer to establish total order for a batch of records. This design enables one to add or remove shards without downtime, allows clients to choose the location of records, and enhances scalability [29].

Unfortunately, the design choices that enable these desirable properties in today's shared logs fundamentally lead to high *delivery* latency (§2). Delivery latency refers to the time between when upstream components produce records and the *earliest point* when the shared log can deliver these records to downstream tasks. The key reason for high delivery latency is the overhead to determine the global order and doing so in a batched manner. Specifically, the shards periodically batch and report the number of records made durable so far to the sequencing layer; the sequencing layer then determines a "global cut", which dictates how to assign positions to durable records across shards. Only after the shards coordinate with the sequencing layer and receive the global cut, can they assign positions to their records and serve them to downstream tasks, increasing delivery latency.

Such high delivery latency, in turn, leads to high application *end-to-end (e2e)* latency, which refers to the time between when records are produced and when the downstream task completes processing them. This is because only after the shared log orders and delivers records, can the application's downstream computations start, increasing e2e latency. Realizing low e2e latency, however, is critical for many real-world applications. Consider the following: a financial application that must flag fraudulent activity as soon as transactions are ingested [25, 62]; real-time analytics that must derive insights from new data as early as possible [6, 56]; high-speed trading that must trade quickly based on market feed [28, 34]. All these applications desire low e2e latency in addition to elasticity, flexible placement, and scalability from the shared log. Although today's shared logs like Scalog satisfy the latter needs, they cannot help realize low e2e latency.

SpecLog Abstraction and Fix-Ante Ordering. To enable applications atop shared logs realize low e2e latency, we identify an opportunity for speculative execution. Given that high delivery latency stems from global ordering, we observe that if a shared log can *speculate* the global order and deliver

records quickly, downstream computation can start based on the speculated order. Thus, the shared log's coordination and the downstream application's computation can be overlapped, reducing e2e latency. This approach suits real-world applications because they typically perform computation over the consumed records (e.g., computing aggregates [26], updating indexes [14, 19], performing analysis [31]).

Based on this observation, we propose SpecLog, a new shared-log abstraction (§3.1). SpecLog offers the same interface as current shared logs but with one difference. SpecLog first delivers records in a predicted order, and later confirms whether or not the prediction is correct. If correct, SpecLog enables low e2e latency; otherwise, SpecLog notifies applications to recompute based on the actual order, preserving correctness. While SpecLog's idea can benefit any shared-log design, our focus is on durability-first architectures that offer elasticity, flexibility, and scalability which are critical to applications in addition to low e2e latency. In such a durability-first SpecLog system, shards operate the same way as today's logs, periodically coordinating with the sequencing layer to determine the actual order. However, they predict the global order and deliver records earlier before coordination. Later, when the actual order arrives, shards confirm or fail the speculation.

Using speculation to reduce latency is *not* new; many prior distributed systems have done so [23, 33, 38, 39, 48, 65]. However, SpecLog is the first to offer a speculative interface for today's shared logs, helping applications realize low e2e latency while preserving the other properties of today's shared logs. Further, a key innovation in SpecLog is how it virtually eliminates misspeculations except in very rare cases via a novel ordering mechanism called *fix-ante* ordering.

To realize low e2e latency, SpecLog must be able to correctly predict record positions at the shards before global coordination. Such correct prediction in current logs like Scalog, however, is inherently difficult. This is because shards have the *free will* to make durable and report however many records they wish in each report to the sequencing layer. Because the positions assigned to records at a shard depend upon how many records *other* shards report, it is hard for the shard to predict where its local records will sit in the total order.

Fix-ante[†] ordering solves this problem (§3.2). The key idea behind fix-ante ordering is to *predetermine* beforehand the global order and try to enforce the shards to *adhere* to this order. Fix-ante order is specified as a series of predetermined global cuts. The shards adhere to each predetermined cut by making durable and reporting exactly the number of records as dictated by that cut (which we refer to as the shards' *quotas*). Because each shard knows that other shards will report precisely the number of records dictated by their quotas, each shard can accurately predict the positions of its local records in the total order without waiting for the actual global order (which is determined after coordinating with the sequencer).

[†]Fix-ante is derived from a Latin phrase "Praefixum ante" which roughly means to fix or predetermine beforehand (as opposed to having free-will).

Note that fix-ante ordering does *not* mean order is established before durability; rather, fix-ante ordering simply offers a way for predicting the global order in durability-first architectures.

As long as the shards adhere to their quotas, the actual global order will match the fix-ante one, ensuring that the positions speculated by the shards remain correct. Fortunately, SpecLog can ensure that shards can adhere to their quotas except in rare cases. Specifically, in failure-free cases, shards can adhere to their quotas even when they have fewer or more records than their quotas by filling in *no-op* records or by delaying records to subsequent reports, respectively. Even when a few shard replicas fail, quotas can be adhered to because shards are internally fault tolerant. Only in the rare case where an entire shard fails or if no shard replica can contact the sequencing layer, will the shard be unable to adhere to its quota, resulting in incorrect predictions. While SpecLog loses some performance under such rare cases, it still preserves correctness by failing the speculation.

Belfast System. We build Belfast, an implementation of the SpecLog abstraction and fix-ante ordering (§4). While fix-ante ordering provides a basic framework for accurate predictions, a real system must solve several performance challenges while still preventing misspeculations. First, Belfast must assign quota for a shard such that it is neither too high (which will add many no-ops) nor too low (which will delay records); Belfast achieves this by setting quotas based on ingestion rates. Second, even with the right quotas, Belfast must handle cases where shards experience bursts. Belfast handles bursts via a new *lag-fix* mechanism; without this, appends and confirmations will incur higher latencies. Third, Belfast must handle longer-term rate changes. Belfast does this by dynamically adjusting predetermined cuts and quotas. Importantly, it does so without misspeculations via a novel *speculation lease window* technique that ensures that all shards use the same predetermined cuts for predictions and that they uniformly move to the new cuts at the window boundaries. Belfast also adds or removes shards without downtime and misspeculations using the speculation lease windows. Finally, Belfast can dynamically exclude shards from some cuts to mitigate straggler shards and to maintain low latencies with many shards.

Results. Our experiments (§6) show that Belfast delivers records ∼3× earlier than Scalog, thereby reducing e2e latency by 1.6× over Scalog. Belfast offers significant benefits in e2e latency even with varying amounts of downstream computation. Belfast achieves low e2e latency while incurring minimal overhead in append latency over Scalog (5.8% with 10 shards). We also show that Belfast can effectively handle bursts, adjust to rate changes, mitigate straggler shards, and handle shard failures, all while maintaining low e2e latency. We show that, like Scalog, Belfast can add and remove shards without any downtime and scale throughput with shards while reducing e2e latencies. Finally, we build three applications (§5): intrusion detection, fraud monitoring, and high-frequency trading, and show that Belfast enables these applications to realize

1.4×-1.6× lower e2e latency than Scalog.

Contributions. This paper makes four contributions.

- We propose SpecLog, a shared-log abstraction that delivers records by predicting their order, overlapping global coordination and computation, thus reducing e2e latency.
- We introduce fix-ante ordering, a novel ordering approach that enables shards to accurately speculate global positions.
- We build Belfast, an implementation of SpecLog and fix-ante ordering. Belfast is the first shared log to offer elasticity, flexible placement, and scalability with low e2e latencies.
- We experimentally demonstrate Belfast's benefits.

2 Motivation

We explain why today's shared logs incur high delivery latencies, making them unsuitable for realizing low e2e latency.

2.1 State-of-the-Art Shared Logs: Background

Shared logs [16–18, 29] offer a powerful abstraction: a fault-tolerant, ordered record sequence that many clients can simultaneously append to and read from. Upon appends, the shared log makes records durable and linearizably [35] orders them: if an append *A* completes before another append *B* starts, then *A* will precede *B* in the log. Clients can read records at positions or subscribe to receive records in order.

With this abstraction and simple interface, shared logs ease building modern applications like real-time analytics [52, 54, 59, 60], high-frequency trading [27, 28], and many others [15, 31, 37, 46, 49, 51]. In all these applications, a set of upstream components ingest data into the shared log which provides strong durability and ordering guarantees. Downstream tasks then consume the records and compute over them.

Popular shared logs like Kafka [13] and others [5, 50] do not provide a total order across shards. Corfu [16, 17], an early shared log, offers total order. Corfu adopts an *order-first* approach, where clients first obtain a position from a sequencer. Then, using a fixed position-to-shard mapping, clients write the record to the shard responsible for the obtained position. As others have noted [29, §2.2], the order-first approach is harmful in three ways. First, it introduces holes because clients can fail after getting the position but before writing. Readers must fill such holes, which requires a system-wide mapping from positions to where records are stored, and clients and servers must agree upon this mapping. This means adding or removing shards requires agreeing upon the new mapping, which leads to system-wide unavailability. Second, the fixed mapping also means that clients cannot flexibly write records to shards of their choice. Finally, obtaining a position for every write makes the sequencer a scalability bottleneck.

State-of-the-art shared logs like Scalog [29] address these limitations. Scalog takes a *durability-first* approach, where clients first store records on a shard of their choice. After batching many records, the shards contact a sequencing layer which determines the total order of records across shards. By avoiding Corfu's fixed mapping, Scalog can add or remove

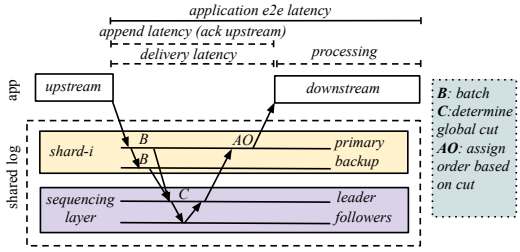


Figure 1: **Delivery and E2E Latency in Today's Shared Logs.**

shards without downtime, enabling it to elastically grow or shrink throughput and capacity [29, §6.1]; such seamless elasticity is important for applications to adapt to dynamic demands [29, 41]. Scalog also enables flexible data placement as clients can choose record locations; this is greatly useful because applications can assign records to shards based on application-level semantics (e.g., records of a particular user can go to one shard) or based on proximity. Finally, the batched communication with the sequencing layer allows Scalog to scale better than Corfu [29, §6.3]; such scalability is important for applications that must ingest and process data at high rates [29, 34]. Given these benefits, many systems like Boki [36] and FlexLog [32] adopt a Scalog-like design.

2.2 High Delivery Latency in Today's Shared Logs

Although today's shared logs offer desirable properties, the design choices that enable them to achieve these properties is fundamentally at odds with low *delivery* latency, which is the time between when records are produced into the log and the earliest point they can be delivered to downstream tasks.

Figure 1 shows this problem in Scalog. Clients first send records to the primary of a shard. The shard primary logs the records locally and replicates them to the shard backup. A record is durable once it has been logged by both the shard primary and backup. The shard servers then report their local log lengths to a sequencing layer. To amortize the cost of contacting the sequencing layer, the shards batch (***B*** in Figure 1) and only report log lengths periodically; such periodic reports incur a batching delay. The sequencing layer then assembles the local reports from shards and determines the number of durable records within each shard. It then computes a *global cut*: the global order of durable records across shards (***C***), and makes the cut fault-tolerant via Paxos [40]. The sequencer then sends the global cut to all shards, using which each shard assigns global positions to records in its local log (***AO***). Only after this point, in addition to acknowledging the appends, the shared log can deliver the records to downstream consumers, which then can start processing. Thus, between upstream ingestion and downstream consumption, applications incur batching delays and the cost of global ordering.

High delivery latencies are not merely a result of how today's shared logs are implemented, but rather a fundamental cost of the durability-first approach. This is because the design choice to have clients first write records to shards of their choice and then have the shards report to the sequencing layer

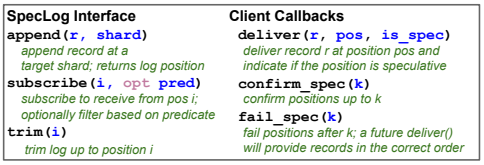


Figure 2: **SpecLog Interface.**

in a batched manner (which enables seamless elasticity, flexibility, and scalability) is what leads to high delivery latencies. Order-first shared logs like Corfu do not incur some of the above overheads like batching and coordination within the sequencing layer. However, they also deliver records only after the global order is established which takes multiple roundtrips [16, 43], thus still incurring high delivery latencies (besides suffering from other limitations for applications).

Unfortunately, high delivery latencies negatively impact application e2e latency. As shown in the figure, e2e latency includes the delivery latency and the time to process the records. Thus, to applications, the high delivery latencies of today's shared logs is a major obstacle to realizing low e2e latencies.

2.3 Demand for Low E2E Latency

Low e2e latencies are critical for many modern, data-driven applications. High-speed trading applications must execute trades in real-time based on incoming data feeds [27, 28, 34, 42], where shaving off even *milliseconds* of latency can save millions of dollars [9, 10]. Financial fraud detection systems need to flag suspicious transactions as quickly as possible [25, 55, 62], with every millisecond being crucial [2, 12]. Real-time analytics engines [52, 59] require rapid insights from fresh data, while real-time search systems must minimize the delay before new data appears in results [54, 60]. Similar low e2e latency requirements exist for event sourcing [37, 51], IoT analytics [24, 31, 49], and fleet management applications [58]. Unfortunately, current shared logs cannot meet these demanding latency requirements. Further, the importance of low e2e latency and thus the need for low delivery latency is evident from RedPanda's recent survey [11], where 100s of practitioners reported the vital metrics in their shared-log deployments. Among them, ∼35% cited delivery as the most critical latency metric.

3 SpecLog and Fix-Ante Ordering

We describe the SpecLog abstraction and explain how it uses fix-ante ordering to enable accurate speculations.

3.1 SpecLog Abstraction

Our goal is to design a shared log that enables low e2e latency. To this end, we first observe that in many modern, data-driven applications, downstream tasks typically perform computation over the consumed records. For example, IoT analytics performs aggregations, detects outliers, updates database views, or runs ML algorithms [20, 26, 49]. Real-time search platforms [54] incrementally update multiple indexes (e.g., document, columnar) as new data is ingested [14, 19]. Similarly, high-speed trading computes online aggregations and runs regression to predict expected profit and loss [42].

Current shared logs incur high delivery latencies and only after this delay, can downstream computations start, increasing e2e latencies. Given that high delivery latency stems from global ordering, we observe that if the shared log predicts global positions and delivers records earlier, downstream computation can start based on the speculated order. This allows the shared log's global coordination and downstream computation to be overlapped. That is, by the time the processing completes, the shared log could have determined the actual order after global coordination. If the predicted order matches the actual one, then applications realize low e2e latency.

Based on this observation, we propose the SpecLog abstraction. SpecLog offers a similar interface to today's shared logs [29] as shown in Figure 2. Applications can append records at a shard of their choice, subscribe[†] to records starting from a position (optionally filtering based on a predicate), and trim the log. The only difference is that when a record is delivered for a position, a bit indicates whether or not the position is speculative. If speculative, SpecLog will later confirm or fail the speculation. Applications require little changes to work with SpecLog's interface. First, applications must wait until they receive the confirmation before externalizing outputs (e.g., trigger alerts). Next, if downstream tasks perform state updates, they must roll them back if speculation fails.

SpecLog's idea can benefit any shared-log architecture, even order-first ones. However, our focus is on today's durability-first logs that offer elasticity, flexibility, and scalability because applications require these properties in addition to low e2e latency. Figure 3 shows how SpecLog would reduce e2e latencies in such durability-first shared logs. In SpecLog, once the records are durable, the shards predict their order (***P***) and deliver them to downstream tasks, which begin processing. Shards continue to process appends the same way as existing logs: they coordinate with the sequencing layer in a batched manner (***B***) to determine the actual order. Once the actual global cut is determined (***C***), SpecLog checks if the predicted order matches the actual one (***CO***). If the prediction is correct, SpecLog sends a confirmation; otherwise, the speculation fails and records are supplied in the correct order.

Properties. SpecLog reduces e2e latency compared to today's shared logs regardless of compute time. However, the extent of benefit depends upon how big or small the compute time is. Intuitively, SpecLog offers the most benefit when global ordering and computation effectively overlap; in other cases, SpecLog still offers lower e2e latencies. SpecLog preserves the seamless elasticity, flexibility, and scalability of today's shared logs. First, in SpecLog, clients still flexibly choose shards to write records. This, in turn, allows SpecLog to add or remove shards seamlessly without any downtime [29]. Finally,

[†]SpecLog can also support a read API, but the applications we target prefer subscribe because it enables them to consume records at the earliest.

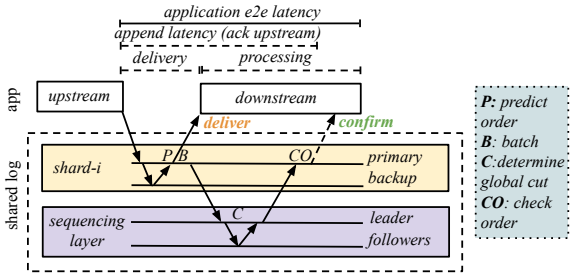


Figure 3: **Delivery and E2E Latency in SpecLog.**

because appends are batched at the shards which then contact the sequencing layer periodically, SpecLog would scale well.

3.2 Fix-Ante Ordering

SpecLog can lead to low e2e latencies only when the shards can correctly predict the position of their records. Unfortunately, making such correct predictions is difficult in durability-first shared-log designs. We observe that the *free-will* nature of reports is what makes correct predictions hard.

To understand this, consider Scalog with three shards S_1, S_2, and S_3. Suppose, initially, S_1, S_2, and S_3 report 2, 3, and 2 durable records to the sequencing layer, respectively. As a result, the sequencing layer would produce the global cut $\langle 2,3,2\rangle$. Scalog uses a deterministic, lexicographic scheme to assign order, where records in lower-numbered shards go before records in higher-numbered shards [29]. Thus, from the global cut, shards would assign positions to their local records as $S_1 : [1,2], S_2 : [3,4,5], S_3 : [6,7]$. One approach to predict the positions for subsequent records is to use the information from the previous cut (that S_1, S_2, and S_3 sent 2, 3, and 2 records, respectively). If such a scheme is used, S_2 would predict that its next local record would get position 10 (because it anticipates that S_1 would report 2 records and thus positions 8 and 9 would go to S_1). However, in the next report, S_1 *could* report 3 records (because it has the free-will to do so); thus, S_2's next local record will actually be assigned position 11 (not 10), making S_2's prediction incorrect.

Fix-ante Ordering Insight. Fix-ante ordering, in contrast, enables shards to accurately predict the positions of their records. The main idea behind fix-ante ordering is to *predetermine* in advance the global order and make the shards *adhere* to this order. Fix-ante order is a sequence of predetermined global cuts that the system is expected to produce. Each cut dictates how many records shards must include in their reports, which we refer to as the shards' quotas. Shards then adhere by exactly meeting their quotas in each report. The key insight is that, because each shard knows that the other shards will report precisely the number of records dictated by their quotas, each shard can accurately predict the positions of its records in the total order before global coordination. We make this idea more concrete below.

Predetermined Cuts, Quotas, and Predictions. The fix-ante order is specified as a sequence of predetermined global cuts $P_1, P_2, P_3, P_4 \ldots$ that the shards know beforehand. The i^{th} predetermined cut is given by $P_i : \langle d_{i1}, d_{i2}, d_{i3} \ldots d_{in}\rangle$, where d_{ij} is the expected number of durable records at shard j as of the i^{th} cut and n is the number of shards. The quota for a shard j to satisfy the i^{th} predetermined cut, q_{ij}, is the exact amount of new records that shard j must make durable and include in its i^{th} report. That is, $q_{ij} = d_{ij} - d_{(i-1)j}$.

Given this, the shards predict positions of their local records in the total order as follows. Intuitively, the positions for the j^{th} shard for the i^{th} cut will start after all the positions covered by the $(i-1)^{th}$ cut and the quotas for all the shards that precede j in the i^{th} cut[†]. Precisely, the predicted positions for the j^{th} shard for the i^{th} cut will be $[S+1, \ldots, E]$, where $S = \sum_{k=1}^{n} d_{(i-1)k} + \sum_{k<j} q_{ik}$ and $E = S + q_{ij}$.

As an example, consider the predetermined cuts for three shards S_1, S_2, and S_3 to be $\langle 1,2,2\rangle$, $\langle 2,4,4\rangle$, $\langle 3,6,6\rangle$ and so on. In this simple example, the quotas remain constant for all cuts with S_1, S_2, and S_3 having a quota of 1, 2, and 2, respectively. Given this, S_1, S_2, and S_3 would predict their record positions for the first cut to be $[1]$, $[2,3]$, and $[4,5]$, respectively. For the second cut, they would predict the positions of their subsequent records as $[6]$, $[7,8]$, and $[9,10]$, and so on.

Although SpecLog shards predict the positions based on fix-ante order, it does *not* mean that order comes before durability in fix-ante ordering. The system still determines the *actual* order by coordinating with the sequencing layer after durability. Determining the actual order is required because, in rare cases (as we describe below), shards may not be able to meet their quotas. To know the actual order, like in other durability-first logs, SpecLog shards periodically report to the sequencing layer the number of records they have made durable. The sequencing layer then sends the actual cut. As long as the shards adhere to their quotas, the actual cuts sent by the sequencing layer will match the predetermined cuts.

Adhering to Quotas and Fix-ante Order. Can shards always adhere to quotas? Fortunately, shards in SpecLog can adhere to their quotas except in rare cases. During normal operation, three cases are possible when a shard tries to meet its quota. First, the shard could have received from clients and made durable precisely the number of records as its quota; in this case, the shard naturally meets the quota and so simply reports all records. Second, a shard could have received and made durable fewer records than its quota. In this case, it adds *no-op* records to satisfy the quota; no-op records are ignored by downstream consumers. Third, a shard could have made durable more records than its quota. In this case, the shard delays the additional records to the next report.

Even when a few shard replicas fail, the shard can keeping meeting its quota; this is because shards are fault tolerant [16, 29, 43] as they internally use primary-backup replication [21] or Paxos [40]. Only in the very rare case where an entire shard fails or if no replica within the shard can contact the sequencing layer, the shard will not be able to adhere to

[†]This is because, SpecLog uses the same lexicographic procedure as Scalog to assign positions across shards from cuts.

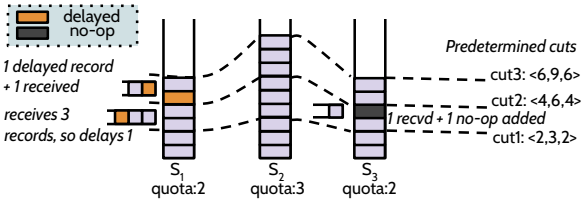


Figure 4: **Adhering to the Fix-ante Ordering.**

its quota and thus the actual cut will be different from the predetermined cut, making the predictions incorrect. While SpecLog loses some performance under such cases, it still preserves correctness by failing the speculation.

Figure 4 shows how shards can meet their quotas, which enables correct speculations. In this example, the predetermined cuts are $\langle 2,3,2\rangle$, $\langle 4,6,4\rangle$, $\langle 6,9,6\rangle$ and so on, and the shards S_1, S_2, and S_3 have a constant quota of 2, 3, and 2, respectively. As shown in the figure, initially, S_1, S_2, and S_3 have received and made durable 2, 3, and 2 records, respectively, allowing them to meet the quota by simply reporting all records. Next, S_1, S_2, and S_3 have received 3, 2, and 1 records, respectively. To meet its quota, S_1 delays one record to the next report (the orange record); similarly, S_3 adheres to its quota by adding one no-op record (the gray record). S_2 can thus accurately predict that its next local record will be assigned a global position of 10 because it knows that positions 8 and 9 would go to S_1 based on the predetermined cuts. This is in contrast to the free-will nature of reports in today's shared log, which makes correct predictions hard (as we described earlier).

While each shard adheres to its quota, the sequencing layer waits to receive reports from all the shards to ensure that the entire system followed the fix-ante order. Only when it knows that they have collectively adhered to the predetermined cut P_i; can the sequencer send the corresponding actual cut A_i. In the above example, suppose in the second report, the sequencer only received reports from S_1 and S_3 and has not heard from S_2. Then, it cannot send the second actual cut. If it sends the actual cut now, it would be $\langle 4,3,4\rangle$ instead of the predetermined cut $\langle 4,6,4\rangle$, which would make the predictions wrong. Thus, to match the predetermined cut, the sequencer waits until it knows that the shards have adhered to their quotas. Note that if any shards have a quota of 0 for a predetermined cut, then the sequencing layer does not wait for those shards. If the sequencer finds that it did not receive a report from a shard for a long duration (e.g., due to a failure), the sequencing layer will determine the actual cut based on the received reports; in such a case, the actual cut will be different from the predetermined one, which will be handled as a misspeculation.

Append Linearizability. With fix-ante order, although the cuts are predetermined, appends still *cannot* be acknowledged without coordinating with the sequencing layer; acknowledging appends before that will violate linearizability. Consider a scenario with two shards S_1 and S_2, and the first predetermined cut $\langle 1,1\rangle$. Suppose an append B happens at shard S_2

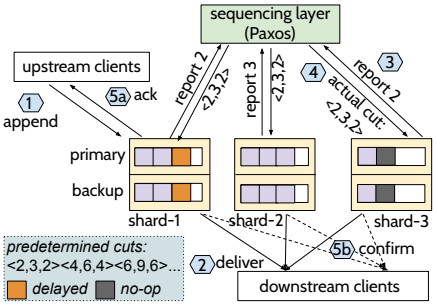


Figure 5: **Belfast Overview, Append and Delivery Path.**

(which will be assigned position 2 according to the predetermined cut). Assume that the system *incorrectly* acknowledges the append right away. Now, it is possible for another append A to start after this and be written to S_1, which will be assigned global position 1 (again according to the cut). Such ordering violates linearizability because A follows B in real time but is assigned a position before B in the total order. Thus, even with predetermined cuts, SpecLog waits for the actual cuts from the sequencing layer before acknowledging appends. In the above example, append B will be acknowledged only after receiving the actual global cut from the sequencing layer; the same cut will also cover append A, essentially making A and B concurrent, thus preserving linearizability.

Implication of Waiting to Satisfy Fix-ante Order. In Scalog, shards send however many records they have made durable at a fixed periodic interval. Also, Scalog's sequencing layer determines and sends the global cut at periodic intervals by including whichever shards have reported in that interval. Fix-ante ordering changes this behavior to keep the predictions correct. First, shards only report after they meet their quotas. Also, the sequencing layer sends the actual global cut after it waits for the shards to adhere to the corresponding predetermined cut. Thus, when shards do not report in a timely manner (e.g., because shards do not have enough records or are simply slow), it delays the actual global cut from the sequencing layer. Thus, fix-ante ordering in its original form (described thus far) will incur overhead in append latencies and the time to confirm the speculations. We next describe how with careful techniques, a system that uses fix-ante ordering, can alleviate this issue.

4 Belfast Design and Implementation

Belfast is an implementation of the SpecLog abstraction and fix-ante ordering. Although fix-ante ordering provides a basic framework for predictions, Belfast must ensure high performance under realistic settings, while enabling accurate predictions. We first describe Belfast's basic operation (§4.1), then how it addresses performance challenges (§4.2-§4.7), and finally how it handles failures (§4.8).

4.1 Basic Operation

Architecture. Figure 5 shows Belfast's architecture. Like Scalog, Belfast has a set of shards, a sequencing layer, and

clients. Each Belfast shard runs standard primary-backup replication [21] and is therefore fault-tolerant. The sequencing layer is a Paxos [40] group that determines actual global cuts and makes them fault-tolerant. Upstream clients can append records to shards of their choice and downstream clients subscribe to receive records from the shards.

Append Path. Figure 5 shows Belfast's append and record delivery path. The append path is similar to that of Scalog but has two differences. Like Scalog, clients first write records to shards, which then make the records durable (step-1). In Scalog, the shards contact the sequencer every *ordering interval* (T_{ord}), which enables them to batch and report many records. Belfast also does the same (step-3). The first difference is that Belfast shards adhere to quotas dictated by the predetermined cuts when reporting to the sequencer; the shards adhere by either reporting all new records, filling in no-ops, or delaying records. The second difference is how Belfast determines the *actual* global cut. In Scalog, the sequencer determines the global cut once roughly every T_{ord}, including the records of whichever shards have reported in that interval. In contrast, Belfast's sequencing layer sends the i^{th} actual global cut only when it knows that the shards have adhered up to the i^{th} predetermined cut (step-4). Finally, the upstream clients are acknowledged (step-5a).

Delivery Path. In Scalog, shards deliver records only after receiving the global cut (i.e., after step-4). In contrast, in Belfast, shards deliver records once they are durable by predicting their positions using the predetermined cuts without waiting to contact the sequencer (step-2). Once a shard receives the actual global cut (after steps 3 and 4), it checks if the actual cut matches the predetermined cut. If so, it sends a confirmation (step 5b); otherwise, it fails the speculation for the delivered records, after which the downstream clients would rollback and recompute based on the correct order.

4.2 Rate-based Quotas

A key factor that affects Belfast's performance is the shards' quotas. A shard's quota must be "right-sized". If too high, many no-ops will be added leading to extra work; if too low, many records will be delayed to subsequent reports, increasing the append and confirmation latencies for delayed records.

Belfast addresses this problem by predetermining cuts in such a way that the quotas match the shards' ingestion rates. With such rate-based quotas, each shard can naturally meet its quota in every report. That is, a shard will receive from clients and make durable roughly the same number of records as its quota every T_{ord}. However, even with a stable ingestion rate, there could be variations and thus the shard could have received and made durable more or fewer records than its quota. Belfast handles this by delaying records to subsequent reports or filling no-ops.

Rate-based quotas work well when rates are roughly stable and if shards are never added or removed. However, in reality, shards could experience bursts or drops in rates. The rates could also increase or decrease more permanently. Shards could be added or removed to meet dynamic application demands. Finally, a shard could be straggling, affecting the append and confirmation latencies for other shards. In the rest of this section, we describe how Belfast handles these problems, and importantly, how it does so without any misspeculation, preserving the e2e latency benefits of fix-ante ordering.

4.3 Handling Bursts and Drops via Lag Fix

When a shard experiences a burst, if the additional records in the burst are delayed to the subsequent ordering intervals, then it may take several intervals to drain the burst. This will increase the append and confirmation latencies for those records. Thus, instead of delaying additional records to subsequent intervals, Belfast shards report to the sequencing layer *immediately* once its quota is met. With this, in the normal case, when the rate is stable, a shard would report roughly once per T_{ord}. However, during a burst, the shard will send many reports in one ordering interval, draining the burst.

Such high-frequency reports alone, however, do not ensure that the requests in the burst can be acknowledged. This is because other shards would still be reporting once every T_{ord}, satisfying one predetermined cut every T_{ord}. Thus, the sequencing layer cannot send the actual cuts for the additional reports sent by the bursty shard. For example, if a bursty shard (say S_1) sends five reports while another shard (S_2) has sent one report, the sequencer can only send the first actual cut and cannot yet send the actual cuts for S_1's last four reports, delaying appends and confirmations of records in the burst.

To reduce the append and confirmation latencies, Belfast uses a *lag-fix* technique. Belfast first detects the burst at the sequencing layer by observing the high frequency of reports from a shard. It then realizes that the other shards are "lagging" and thus asks them to send more reports. The lagging shards then do so with client requests that they have and additionally filling no-ops as needed. This enables Belfast's sequencer to send the subsequent actual cuts. In the above example, once the sequencing layer informs S_2 about the lag and S_2 sends four additional reports, the next four actual cuts can be sent.

The lag-fix mechanism also helps Belfast handle sudden drops in ingestion rates. When a shard experiences a drop, it cannot meet its quota naturally. Although the shard will fill no-ops to meet the quota, doing so takes time because Belfast is conservative about filling no-ops (to minimize its overhead); this can result in delayed reports. Meanwhile, the other shards could have progressed more, causing this shard to lag behind. But fortunately, the lag-fix mechanism would instruct the shard to fill in no-ops immediately to satisfy many subsequent cuts and report them in one go; this helps the lagging shard quickly catch up with the rest of the system.

4.4 Speculation Lease Windows

The ingestion rate at a shard might change more permanently (for example, as clients join or leave). The mechanisms de-

scribed so far help Belfast handle slight variations and bursts. However, relying on them for long-term change will lead to performance overhead. In particular, if a long-term rate increase is handled by just high-frequency reports and lag fix, then it will introduce too many no-ops from the lagging shards; the sustained high frequency of reports will also increase the load on the sequencing layer. Similarly, a long-term rate decrease cannot just be handled by filling no-ops.

Thus, the system must periodically adjust predetermined cuts and quotas. Belfast's sequencing layer acts as a convenient place to do this because it can estimate the rates based on the frequency of reports. Further, since the shards periodically contact the sequencing layer, it can conveniently inform the shards of any changes in the cuts and quotas.

The key challenge is to change cuts without misspeculations. In particular, we must avoid cases where some shards use the old predetermined cuts, while others use the new cuts; if not misspeculations can arise. Suppose the current quotas for three shards S_1, S_2, and S_3 are 2, 3, and 2, respectively. Suppose the rate at S_1 increases, requiring a quota of 4. Assume the sequencer sets S_1's quota to 4 immediately after determining the first actual cut and sends the updated quota to the shards. However, based on the previous predetermined cuts, shard S_2 might have already delivered records for positions 10, 11, and 12 speculatively. But, with the new quota, S_1 would assign positions 8, 9, 10, and 11 to its records, which would lead to misspeculations in the predicted positions for S_2's records.

To avoid this problem, Belfast predetermines cuts and quotas for a *window*. All shards use the same predetermined cuts to make predictions within the window. Conceptually, the window is a *lease* on how long the shards can and must use the current predetermined cuts to make predictions. Belfast allows changes to predetermined cuts and quotas only at window boundaries. Thus, all shards uniformly use the new cuts and quotas after the current window ends.

Belfast realizes the speculation lease windows as follows. First, instead of time-based lease windows, Belfast measures windows in terms of number of predetermined cuts. That is, W cuts form a lease window. Shards can only speculate positions covered by W cuts. When the previous window nears completion, the sequencing layer informs shards of the predetermined cuts and quotas for the next window. In the previous example, S_1, S_2, and S_3, would move to the new cuts only when the current window ends. Thus, S_1 will not use the new quota of 4 in the current window and will do so only from the next window, preventing the above misspeculation.

Belfast determines the need for a quota change and the new quota as follows. When the ingestion rate increases (or decreases), a shard will report more (or less) frequently to the sequencing layer. The sequencing layer uses this change in frequency to detect the need for a quota change. Belfast decides the new quota in such a way that every shard, after the quota change, would report roughly once every T_{ord}.

4.5 Adding and Removing Shards Seamlessly

To meet the dynamic needs of applications, Belfast must be able to add and remove shards, and do so without misspeculations. The speculation window mechanism described above provides a good substrate to achieve this goal. The idea is that shards can be added or removed only at the speculation window boundaries. The sequencing layer decides the new quota assignments in the new window and the predetermined cuts will now include the newly added shard or exclude the removed shard. The only difference from Scalog's seamless elasticity is that shards must wait till the current speculation window to end to join or leave. However, window sizes are sufficiently small, such operations need not wait for long.

In Belfast, when a shard wishes to leave or join, it sends a request to the sequencing layer. A leaving shard waits until the current window is over and keeps meeting its quota until then. A joining shard can demand a quota, which the sequencing layer uses to determine the cuts for the new window with the shard included; eventually, the quota is adjusted appropriately.

4.6 Mitigating Straggler Shards

Belfast's mechanisms can help lagging shards catch up with the rest of the system. However, a shard could be slow for reasons other than it not being able to meet its quota. For example, the network could delay the reports or the reporting thread could be de-scheduled for a long time. Belfast must handle such straggler shards; otherwise appends and confirmations for records at other shards will be delayed.

Belfast mitigates straggler shards by first detecting them at the sequencer and then assigning their quotas to be zero in the next speculation window, similar to the quota-change mechanism. In the new window, the other shards can make progress without waiting for the straggler. In the next window, the straggler shard is given a small non-zero quota to see if it can meet it in a timely manner. If it is able to meet, the sequencer can ramp up the quota; otherwise, the shard is not given quotas for the subsequent windows.

4.7 Reducing Latencies with Many Shards

Rate-based quotas, no-op filling, and lag fixes help reduce append and confirmation latencies despite waiting for all shards to adhere to a predetermined cut. However, with a large number of shards, if Belfast assigns quotas for all shards in all cuts, then the possibility of actual cuts getting delayed because of a few slow shards becomes high.

However, the basic fix-ante ordering framework does *not* necessitate that all shards must report for all cuts. That is, the predetermined cuts could be set such that some shards don't have to report records for a cut to be satisfied (i.e., their quota can be 0 in that cut). The sequencer waits only for reports from those shards that have a non-zero quota before it sends the actual cut. Belfast leverages this fundamental property of fix-ante ordering to keep latencies lower with many shards. The idea is to "stagger" the cuts, where the sequencing layer

waits only for reports from a subset of shards to satisfy a cut and waits for different subsets in different cuts.

Belfast's current implementation realizes this idea via a simple strategy, where shards are grouped into g groups. The predetermined cuts are fixed such that the i^{th} cut waits for only shards in group $i \bmod g$. This enables the sequencer to send the actual cuts without waiting for all shards but only a subset for every cut, which reduces append and confirmation latencies. More sophisticated policies can be used to stagger the cuts (e.g., by assigning groups in a more dynamic manner); we leave such policies to future work.

4.8 Failure Handling

Belfast retains Scalog's behavior under sequencing-replica failures: the system is unaffected by follower failures but correctly remains unavailable until re-election if the primary fails. Each storage shard is internally replicated and thus can mask up to f failures. Belfast shards use primary-backup; a consensus group can also be used. Thus, Belfast remains unaffected in the presence of shard-internal failures as well.

Belfast's operation, however, is affected when the shard *as a whole* experiences a failure. This is a very rare scenario in practice, that can occur when more than f replicas within a shard crash or get partitioned, rendering the shard incapable of masking internal failures. Belfast handles whole-shard failures via a *view-change* protocol. Belfast's sequencing layer triggers a view change when it suspects a shard has failed (i.e., the shard is unresponsive to the sequencing layer's lag-fix instruction). During normal operation, Belfast stamps all speculatively delivered records, actual cuts, and confirmations with the view number. For correctness during whole-shard failures, any records speculatively delivered in the old view for which confirmations are pending must be failed.

We call the last position covered by the latest actual cut sent in the old view the *confirmed-gp*; intuitively, records for positions that precede confirmed-gp will not change. On a view change, Belfast fails speculations for all positions after confirmed-gp and informs the clients of the same. Clients rollback their computation up to the confirmed-gp and then re-consume records from the confirmed-gp and recompute.

To avoid re-sending the records from other alive shards or reassigning their positions, Belfast optimizes the above procedure. Instead of excluding a failed shard, S_F, immediately, Belfast fills the log slots that belong to S_F with no-ops for the remainder of the current window. The sequencer assigns this responsibility to an alive shard S_A. With this, speculations for all positions after *confirmed-gp* are failed and clients still rollback. The only change is that the earlier delivered records from other shards and their positions are retained at the clients' buffers; S_A then only delivers no-ops for positions belonging to S_F. S_A also makes durable and includes these no-ops in its reports. This helps all the shards make progress and eventually these positions will be confirmed. In the next window, S_F is excluded and S_A no longer needs to fill no-ops. Note that S_F will stop receiving actual cuts from the sequencing layer and so it will not be able to confirm or acknowledge records for the positions for which S_A filled no-ops. Shard S_A is also responsible for delivering failure notifications and mis-speculations to clients on behalf of S_F. Since all shards are made aware of S_F and S_A from the sequencer through the actual cuts, clients that were only connected to S_F will eventually timeout waiting for confirmations and get re-routed to shard S_A to receive the no-ops and confirmations. The client-side failure handling logic, including the timeout and re-routing, is hidden within the client library.

```
// compute thread state
var lastCompIdx uint64 // last computed log index

func HandleFailSpec(k uint64) {
    pauseCompute()
    for i := lastCompIdx; i > k; i-- {
        undoStateChange(i)
    }
    lastCompIdx = min(k, lastCompIdx)
    resumeCompute()
}
```

Figure 6: **Application Rollback**

4.9 Implementation

We implemented Belfast by modifying open-source Scalog [7]. Our code is publicly available [8]. We introduce only one additional RPC for shards to register with the sequencer and join a Belfast cluster, while all other mechanisms, such as lag-fix and quota change, are realized by piggybacking onto existing RPCs (reports and cuts). Belfast makes all control information (e.g., quotas) also fault tolerant at the sequencer (without extra Paxos rounds). We set the window size to 100 cuts and ordering interval (T_{ord}) to 1ms. We turn on staggered cuts when we have more than 6 shards. To ensure progress, shards begin to fill no-ops to satisfy their quota after $1.5 \times T_{ord}$ since the last report. To minimize the overhead of no-ops, we coalesce consecutive no-ops in the shard-internal log into one on-disk entry and one message for replication; thus, shards can efficiently fill many no-ops.

5 Applications

We built three applications atop SpecLog to demonstrate its viability. These applications run atop conventional shared logs and can be ported to SpecLog's interface with minimal changes. The applications differ in sharding policies, downstream computations, and rollback actions.

IoT Intrusion Detection. Upstream sensors ingest readings into the shared log; different groups of sensors ingest values into different shards. There are many downstream tasks, one per shard. Each task calculates moving statistics, stores them in a local database, and raises alerts if they exceed a threshold value (hinting an intrusion). The cross-shard ordering helps establish the order of intrusion events across sensor groups. With Belfast, a downstream task speculatively calculates the statistics and writes them to a database, but waits for confir-

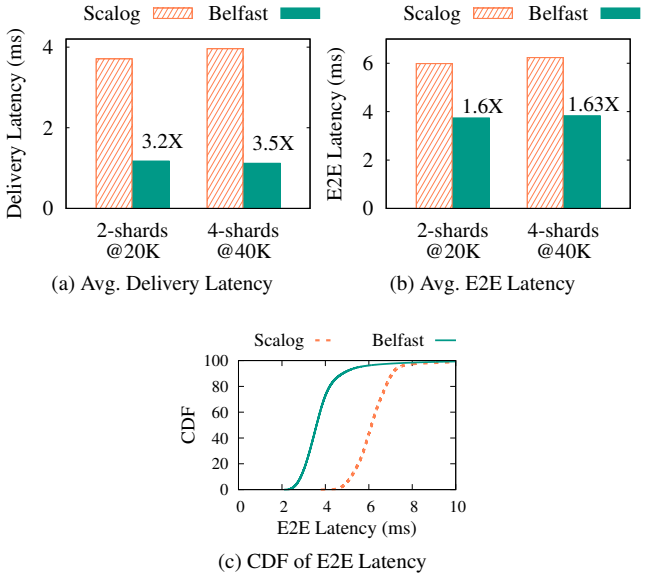
mation before it can raise an alert. If the speculation fails, tasks revert their database writes and reset statistic values.

Fraud Monitoring. This application analyzes users' transactions to detect fraudulent activity. Each transaction is tagged with the originating user's id. Upstream components can ingest transactions into any shard (e.g., the nearest one). There are several downstream tasks, each handling a subset of user-ids. Unlike the previous application, here, each downstream task consumes records from all shards but filters records based on a predicate (user-id). Each task examines if a transaction amount exceeds a threshold to raise alerts. The reader also updates a local database with the running statistics for each user and reverts this state if the speculation fails.

High-Frequency Trading. This application predicts the relationships between stocks based on their prices, which are then used to create pair-trading strategies. Upstream components update the real-time prices of stocks. Appenders can append to any shard or map stock-ids to shards. Downstream tasks subscribe to different stock-ids to receive live price data. Even if incoming data is sharded by stock-id, global ordering across shards is required, as downstream tasks must know the order of price changes across stocks. Downstream tasks predict price relationships via online linear regression. Stock prices are used to update the weights of the model in batches. These intermediate weights are stored in a database to create trading strategies. For rolling back, downstream tasks delete the stored weights, recompute them, and insert the new weights.

Porting Applications to Belfast. For porting applications, Belfast prescribes a few simple guidelines. Firstly, the application must wait for confirmation before externalizing outputs or taking actions. For example, the high-frequency trading application must not execute trades or take decisions based on unconfirmed results. Similarly, the fraud monitoring application cannot flag a transaction as fraudulent until it receives confirmations. Secondly, to deal with mis-speculations, applications have to maintain in-memory state of unconfirmed changes made to the application state in-order to correctly implement rollbacks. This in-memory state is small as there are only a few computed but unconfirmed records at any time. For example, in the fraud-monitoring application, each transaction updates a few rows of the user statistics database, therefore, for each unconfirmed application state change, the in-memory state contains the list of updated rows and their old states.

Implementing these guidelines require minimal changes to the application code for existing shared logs. This primarily includes a thread subscribing to the speculation confirmation or failure callbacks from Belfast and marking unconfirmed results as ready, or performing rollbacks respectively. On receiving a `fail_spec(k)`, the thread pauses compute, undoes the state changes for unconfirmed modifications beyond index `k` and instructs the compute thread to resume from that point onwards. The code snippet in Figure 6 shows how rollbacks are typically implemented with Belfast.



(a) Avg. Delivery Latency (b) Avg. E2E Latency

(c) CDF of E2E Latency

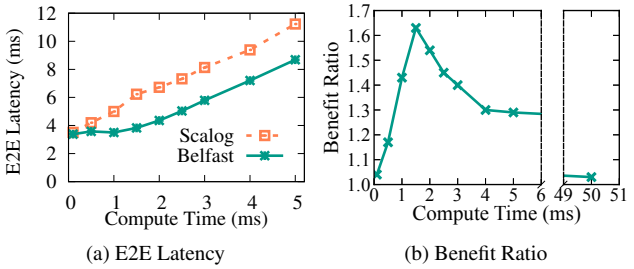
Figure 7: **Delivery and E2E Latency**

6 Evaluation

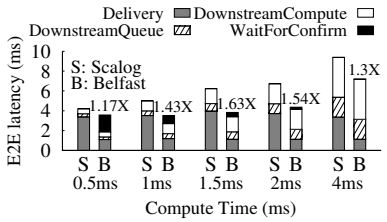
Our evaluation answers the following questions:

- What are the e2e latency benefits of Belfast? (§6.1)
- How do the benefits vary with computation time? (§6.2)
- What are the overheads in append latency? (§6.3)
- How does lag-fix help Belfast with bursts? (§6.4)
- How does Belfast adapt to rate changes? (§6.5)
- How does the speculation lease window size affect Belfast? (§6.6)
- Does Belfast enable seamless elasticity like Scalog? (§6.7)
- Does Belfast scale with shards while maintaining low latencies? (§6.8)
- Does Belfast effectively handle stragglers? (§6.9)
- Do real-world applications benefit from Belfast? (§6.10)
- What is the impact of failures on applications with Belfast? (§6.11)

Setup. We run our experiments on a CloudLab cluster. Each machine has an Intel 10-Core E5-2640v4 CPU, 64GB DRAM, a 25Gb ConnectX-4 NIC, and a SATA SSD. Belfast's sequencing layer has one leader and two followers. Each shard has one primary and one backup. Our main baseline is Scalog; in some experiments, we also compare against variants of Belfast. In Belfast, only the shard primary contacts the sequencer, reporting durable records; further, the reports adhere to the fix-ante ordering. In the original Scalog implementation, both the shard primary and backup contact the sequencer, which then determines which records are durable; also, the shard servers report in a free-will manner. To correctly compare fix-ante ordering and the free-will nature, we modify Scalog so that only the primary reports durable records to the sequencer. In all experiments, we use 4KB records and calculate Belfast's throughput by including only actual records (because clients don't have throughput benefit from no-ops).



(a) E2E Latency (b) Benefit Ratio



(c) Latency Breakdown

Figure 8: **Compute Time vs. E2E Latency Benefits**

6.1 E2E Latency Benefits

We first demonstrate the fundamental benefit of early speculative delivery and fix-ante ordering. We run a workload where upstream clients ingest records and downstream clients consume and compute over them. For a batch of consumed records, computation takes 1.5ms on average. We run with 2 and 4 shards and compare the delivery and e2e latencies of Belfast and Scalog. As shown in Figure 7(a), Belfast delivers records 3.2×-3.5× earlier than Scalog; downstream computation thus can speculatively start. Because of fix-ante ordering, the speculated and the actual order of the delivered records match. Thus, as shown in 7(b), Belfast reduces average e2e latency by 1.6×; we also note improvements in p99 e2e latencies (1.4× and 1.17× with 2 and 4 shards). Figure 7(c) shows the e2e latency CDFs for Scalog and Belfast for 4 shards. In contrast, Scalog delivers records only after global ordering, leading to high delivery and e2e latencies.

6.2 Compute Time vs. E2E Latency

We next examine how compute time impacts e2e latency. We run the same experiment as above with 4 shards but vary the compute time. First, as shown in Figure 8(a), for all compute times, Belfast offers lower e2e latency than Scalog. However, as shown in 8(b), the benefit varies with compute time (e.g., 1.17× with 0.5ms but 1.63× with 1.5ms compute).

We show why benefits vary with compute time by breaking down the e2e latency (8(c)). In Scalog, e2e latency comprises the time to deliver (Delivery), the time records wait in a downstream queue before they are picked for processing (DownstreamQueue), and the compute time (DownstreamCompute). In Belfast, additionally, tasks may have to wait for the order to be confirmed (WaitForConfirm). First, when computation and coordination effectively overlap (e.g., with 1.5ms compute), Belfast offers the most benefit. Second, with shorter compute (e.g., 0.5ms), the benefits decrease, because, although Belfast delivers earlier, once compute finishes, it waits for a while for the order to be confirmed. Next, as compute increases, the wait time for confirmation reduces. However, when the compute is long (4ms), compute exceeds ordering time, reducing benefits. With very high compute times (50ms), the benefit ratio approaches 1. Overall, with all compute times, Belfast offers lower e2e latencies than Scalog; the benefit is maximized when compute and coordination overlap effectively.

6.3 Append Latency

We now analyze Belfast's append latency. We run a workload with a few upstream clients and measure the append latency in Belfast and Scalog with 2, 4, and 10 shards. As shown in Figure 9, Belfast does incur an overhead compared to Scalog. This is due to the shards adhering to the quotas and the sequencing layer waiting to satisfy the predetermined cuts before it sends the actual cuts. However, mechanisms like rate-aware quotas, no-op filling, lag fix, and staggered cuts (for 10 shards as described in §4.9) help Belfast to keep this overhead to a minimum (5.8% with 10 shards). This is a small cost to pay for the significant reduction in e2e latencies, which is critical for many real-time applications (§2.3).

6.4 Handling Bursts with Lag-Fix

We now analyze how Belfast behaves under bursts and the efficacy of lag-fix. We run a workload with an average compute time of 0.8ms with two shards S_1 and S_2, and in the middle, introduce a burst of client requests on S_1 and measure e2e latencies. We compare Belfast against Scalog and a Belfast variant with lag-fix disabled (no-lf). As shown in Figure 10(a), first, Belfast's e2e latency is lower than Scalog throughout. Second, lag-fix generally helps maintain low e2e latencies by constantly fixing lagging shards. Third, lag-fix alleviates the effect of bursts notably. When the burst starts, in no-lf, even though S_1 reports more frequently, S_2 lags behind, delaying confirmations and thus leading to high e2e latencies. The lag persists even after the burst ends. In Belfast, while there is a small increase in latencies when the burst starts (which is also seen in Scalog), lag-fix kicks in, which helps S_2 to catch up and thus reduce latency. Figure 10(b) illustrates the lag-fix behavior by showing the timing of reports in Belfast. Once burst starts, S_1 reports more frequently (blue circles); Belfast detects S_2's lag and asks it to send more reports. S_2 does so (orange squares), resolving the lag.

Dealing with High Burstiness. As seen above, the lag-fix mechanism helps Belfast handle small bursts, drops and variability well. However, repeated bursts could cause many lag-fixes which can increase the load on the sequencing layer and affect client latencies. In such high-burstiness scenarios, Belfast can adopt a couple of techniques to remain performant. One approach is for the sequencing layer to detect such scenarios (based on the frequency of reports) and transition the system into a "high-burstiness" mode, where shards are

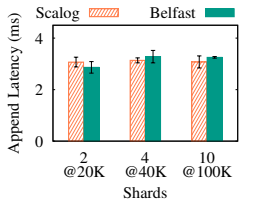
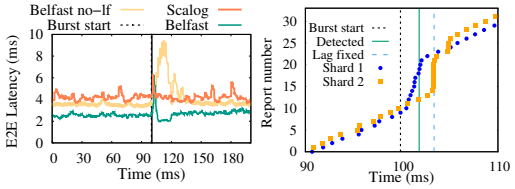
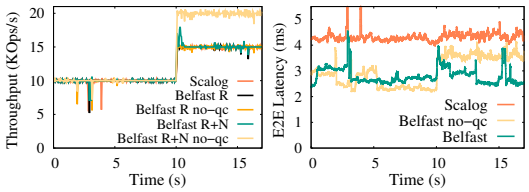


Figure 9: **Append Latencies**



(a) E2E Latency (b) Reports with Lag-Fix

Figure 10: **Lag-Fix Benefits**



(a) Throughput (b) E2ELatency

Figure 11: **Quota Changes**

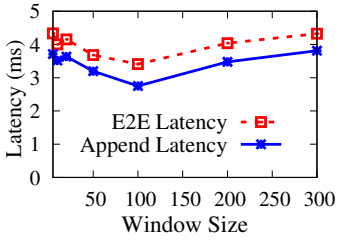
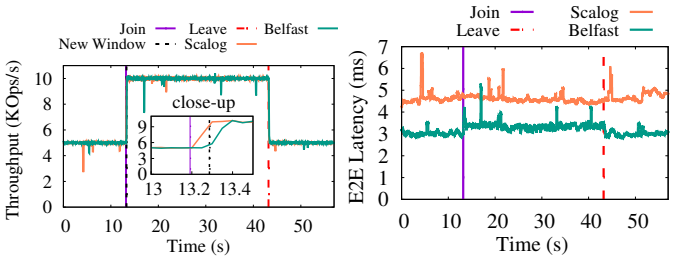


Figure 12: **Lease Window**



(a) Throughput (b) E2E Latency

Figure 13: **Seamless Elasticity**

assigned quotas based on the peak of the bursts and are instructed to *proactively* fill no-ops to match the quota. Under extreme scenarios, Belfast could also dynamically turn-off fix-ante ordering and fall back to Scalog's mode of operation. We leave these approaches as an avenue for future work.

6.5 Changing Quotas Upon Rate Changes

We next examine Belfast's behavior under rate changes using the same setup as §6.4, but in the middle, we add sustained load to S_1, which increases its rate. We compare Belfast to Scalog and a variant with quota-changes disabled but lag-fix enabled (no-qc). Figure 11(a) shows the throughput. For Belfast and no-qc, R+N is the throughput that includes both actual records and no-ops, while R includes only actual records.

Initially, when the rates are stable, no-ops are minimal in both no-qc and Belfast, and they offer the same R and R+N throughput. However, when the rate increases at S_1, in no-qc, lag-fix kicks in, which makes S_2 fill in no-op records. In no-qc, S_1's quota is not changed and it keeps reporting more frequently to drain its "burst". As a result, S_2 keeps filling more no-ops (as can be seen from the higher R+N throughput for no-qc). In contrast, although initially Belfast fills no-ops (due to lag-fix), it quickly adjusts the quota for S_1 and thus S_2 doesn't fill many no-ops; thus, the no-op throughout comes down, keeping R+N close to R[†]. We also noted that in no-qc, the sequencer load was about 2× higher than in Belfast (because of its sustained high-frequency reports). Also, throughout, Belfast's throughput (R) matches Scalog's.

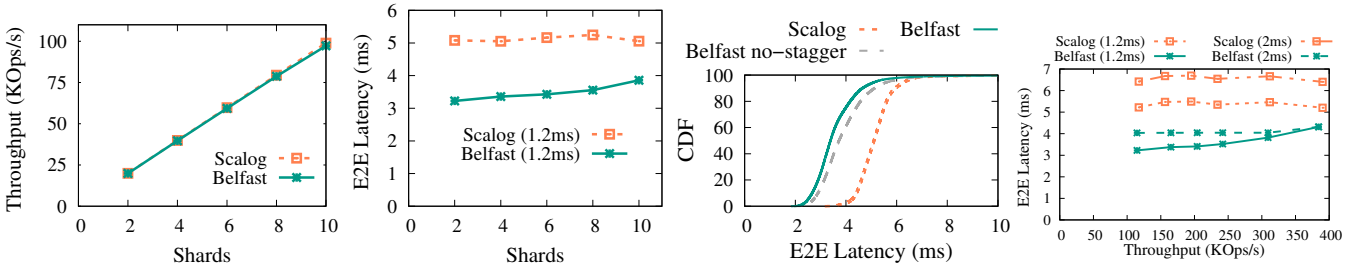
Figure 11(b) shows the e2e latency. Initially, both no-qc and Belfast offer the same e2e latency. However, after the rate change, no-qc suffers from higher e2e latencies because it takes a while to detect the lag and fill no-ops. In contrast, Belfast by changing the quota of S_1 to match its rate, avoids this overhead, leading to lower e2e latencies. Throughout, both Belfast and no-qc offer lower e2e latencies than Scalog.

[†]In all experiments, we noted no-ops added little overhead ($N/R < 5\%$).

6.6 Impact of Speculation Lease Window

We next examine the impact of the speculation lease window size on Belfast. We run a stable workload with 2 shards at 20K throughput at 1.5ms compute with varying sizes for the speculation lease window. As shown in Figure 12, very low window sizes impact both e2e and append latencies. With low window sizes, Belfast incurs frequent synchronization costs where the shards are blocked waiting for the sequencing layer to provide the quota for the next lease window. For instance, we observe that with a window size of 5, the shards spend about an entire ordering interval (T_{ord}) at the end of each lease window waiting for the next quota.

A large lease window size reduces the synchronization cost significantly (∼0.07% of T_{ord} at window size 100), however, Belfast is slower to react. For instance, at 10K throughput per shard, each shard reports 10 records on average every T_{ord}. Due to a transient drop, the ordering layer could assign a quota of 9 to a shard, leaving it in an undesirable state where the incoming rate exceeds the quota assigned for a very large window. This consequently affects client latencies. Furthermore, with a large lease window size, Belfast would also be slower to react to long term rate changes, incurring frequent lagfixes (approximating the behaviour of the no-qc variant in Figure 11). We see that a window size of 100 achieves a good balance and works well for us. In all cases, the e2e latency remains well under the 5.9ms value (see Figure 7(b)) for Scalog under the same workload.



(a) Throughput (b) E2E Latency (c) E2E CDF (10-shards) (d) Emulation Latency vs. Throughput

Figure 14: **Throughput and Latency vs. Shards**

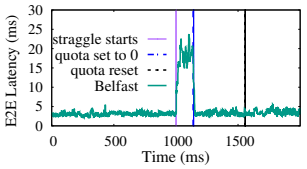


Figure 15: **Straggler Mitigation**

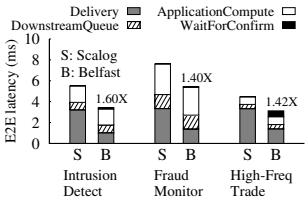


Figure 16: **Applications**

6.7 Seamless Elasticity

We next answer if Belfast offers elasticity without any downtime like Scalog. We also examine if Belfast can do so without causing misspeculations, thus continuing to offer low e2e latency as shards are added and removed. We run an experiment with two shards initially, add two shards on the fly, and then remove them. Figure 13 shows the throughput and e2e latency.

First, as shown in 13(a), when shards are added or removed, Belfast, like Scalog, has no downtime and no drop in throughput. The only difference is that records from the new shards are included only after the current speculation window ends, thus the throughput increase comes a bit late. The close-up shows this: Belfast's throughput increase is delayed but only slightly (< 100ms). Belfast also removes shards without downtime. Next, as shown in 13(b), Belfast offers lower e2e latency than Scalog even as shards are added and removed. This is because Belfast adds or removes shards at window boundaries, which prevents misspeculations, keeping e2e latencies low.

6.8 Throughput and Latency vs. Shards

We next analyze if Belfast can scale throughput with shards like Scalog while offering lower e2e latencies (with 1.2ms compute). As shown in Figure 14(a), Belfast, like Scalog, increases throughput with shards. Next, as shown in 14(b), it does so while offering lower e2e latencies than Scalog (1.66× and 1.4× with 2 and 10 shards). This is also evident from the e2e latency distribution for 10 shards in 14(c). 14(c) also shows the effect of staggered cuts, which helps Belfast to send actual cuts without waiting for all shards; as shown, staggered cuts improve latencies compared to the no-stagger variant.

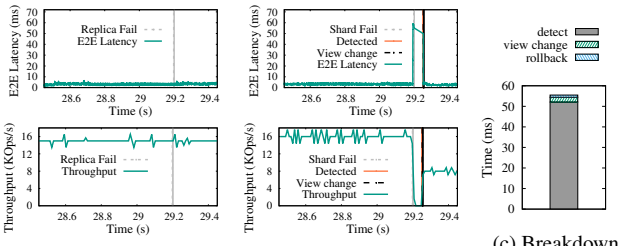
To evaluate scalability beyond 10 shards, we build an emulation framework similar to Scalog [29]. The framework retains the sequencer and shard protocol, but the shards are emulated. Emulated shards report records at the same throughput and latency as real shards but do not receive actual records from clients. Client perceived latency is obtained by adding a delta measured from the real system to latencies measured at the emulated shards. To measure e2e latency, along with the compute, we add the time spent in the downstream queue as obtained in the real experiments. As seen in 14(d), Belfast's throughput scales similarly to Scalog's upto 40 shards. For compute of 1.2ms, Belfast's e2e latencies show a slope as they are limited by the confirmation latencies. For higher compute (2ms), Belfast is at a constant offset from Scalog. In all cases, e2e latency benefits are retained even at 40 shards.

6.9 Mitigating Straggler Shards

We now examine if Belfast can mitigate straggler shards effectively. In the middle of a workload, we inject delays into one shard, which significantly delays its reports. Figure 15 shows the e2e latency. When the shard starts to straggle, e2e latency temporarily goes high. Once the shard repeatedly sends significantly delayed reports, Belfast's sequencer sets the quota of the straggler shard to be zero for the next five windows, and therefore the actual cuts can be sent without waiting for the straggler, which reduces e2e latencies. Eventually, the straggler's quota is reset, and since we remove the injected delay at this point, low e2e latency is maintained.

6.10 Benefits for End Applications

We now evaluate the applications from §5. In all applications, several upstream components ingest data and downstream tasks subscribe to shards according to the application's sharding policy (described in §5) and compute over the consumed records. We run atop Scalog and Belfast, and measure application e2e latencies. As shown in Figure 16, IoT intrusion detection attains the maximum benefit. The breakdown shows that the average compute time for a batch of consumed records in this application is around 1.4ms, which enables almost the entire computation to be overlapped with shared log's coordination. The benefits are less pronounced in fraud monitoring and high-frequency trading which have an average compute time of 2.2ms and 800us, respectively. Thus, in fraud-monitoring, confirmation arrives before computation finishes,



(a) Shard Internal Failure (b) Shard Failure (c) Breakdown

Figure 17: **Application under Failure**

while, in HFT, downstream tasks wait for confirmation.

6.11 Application Behavior under Failures

Sequencing-replica failures do not change the behavior of Belfast compared to Scalog. We thus evaluate the fraud-monitoring application under two shard failure scenarios; shard-replica failures and whole shard failures. We run the shard-replica failure experiment with 3-way replicated shards ($f = 1$) and kill a shard replica in the middle of the run. Figure 17(a) shows that the throughput and e2e latency remain unaffected as the shard internally masks the failure. In the second experiment, we run 2-way replicated shards and fail the *entire shard* in the middle of the run. This scenario triggers misspeculations, which the application handles by performing rollbacks (see §5). As shown in Figure 17(a), e2e latency increases after the shard fails as misspeculations arise. However, once the view is changed and an alive shard fills no-ops on behalf of the failed shard (§4.8), the latency becomes low again. Similar behavior can also be seen with throughput. The breakdown in Figure 17(c) shows that detecting the failure takes up the most time between failure and recovery. More importantly, the time spent by the application to rollback, i.e., undo the state changes and resume is marginal.

7 Related Work

Shared Logs. Corfu [16, 17] cannot seamlessly reconfigure or flexibly place data, and has limited scalability. vCorfu [63] offers flexible placement for readers but still has the other problems. While Scalog [29], Boki [36], and FlexLog [32] address the above problems via a durability-first design, they do not enable low e2e latency, a critical need for applications. Our work is the first to build a shared log that enables low e2e latencies (while preserving the other properties). While we focus on durability-first logs given their desirable properties, Corfu-like logs also suffer from high e2e latencies. SpecLog's idea could enable such logs to offer low e2e latencies as well; specifically, such a system can allow clients to speculatively read ordered but non-durable records from the shard primary.

LazyLog [43], a recent shared log abstraction, reduces *append* latencies by avoiding eager ordering upon appends. However, LazyLog makes a crucial assumption that most reads are decoupled in time from the appends. For the class of applications we target, this assumption does not hold since records are consumed as soon as they are ingested. Furthermore, LazyLog enforces ordering before reads and therefore *does not* allow records to be delivered to downstream consumers before ordering completes. In contrast, SpecLog allows record delivery even before ordering is established and hides this cost by overlapping ordering with downstream computation. As a result, unlike SpecLog, the LazyLog abstraction does not fundamentally help reduce e2e latencies which will always include the cost of ordering.

Prior work [47] reduces subscription latency in pub-sub [30] systems; however, it does not build upon total-order shared logs or reduce e2e latencies.

Speculative Execution. Many distributed systems use speculation in other contexts like consensus [48], replicated stores [33], blockchains [23], BFT [39, 65], distributed file systems [45], transactions [57], atomic broadcast [38], and MapReduce [22]. However, our work is the first to offer a speculative interface for today's total-order shared logs to overlap their expensive coordination with application computation. More importantly, our work shows how to accurately speculate positions via a novel fix-ante ordering scheme.

Pre-partitioning in Consensus. Protocols like Mencius [44, 64] overcome Paxos' leader bottleneck by statically partitioning the consensus log across replicas. Each replica proposes client commands or special *skip* commands (i.e., no-ops) in its slots. While Belfast bears some similarity to these protocols (predetermining slots, filling no-ops), it differs in key ways. First, although Belfast's fix-ante cuts effectively pre-partition the shared log, these cuts are *speculative* in nature (unlike in Mencius). Second, Mencius's pre-partitioning is for replicas within a shard, whereas fix-ante ordering predetermines an expected total order across many shards. Our work also differs in context and purpose: Mencius-like systems pre-partition the log to overcome the leader bottleneck in consensus, whereas Belfast predetermines shared-log slots to enable speculative delivery for low e2e latency in durability-first shared logs.

8 Conclusion

Today's shared logs incur high delivery latency and thus cannot enable low e2e latency, a critical need for applications. SpecLog, a new shared-log abstraction, speculatively delivers records, thus overlapping computation and coordination. Fix-ante ordering enables SpecLog to accurately speculate positions before global ordering. We build Belfast based on these ideas and show that it enables low e2e latencies for applications without sacrificing today's shared logs' properties.

Acknowledgments

We thank our shepherd and the OSDI '25 reviewers for their insightful comments. We also thank the other DASSL members for their discussions and feedback. This material was supported by funding from NSF grants CNS-2340218 and CNS-2339784, an IIDAI grant, as well as a gift from NetApp. We also thank CloudLab [53] for providing a great environment to run our experiments.

References

[1] Apache DistributedLog. https://github.com/apache/distributedlog.

[2] Fighting Fraud in Real-Time: North's Journey to Smarter, Faster ML. https://www.tecton.ai/blog/fighting-fraud-in-real-time-norths-journey-to-smarter-faster-ml/.

[3] Kafka Use Cases - Messaging. https://kafka.apache.org/uses#uses_messaging.

[4] LogDevice: distributed storage for sequential data. https://logdevice.io/.

[5] Pravega: A Reliable Stream Storage System. https://cncf.pravega.io.

[6] Real-Time Analytics Explained. https://rockset.com/real-time-analytics-explained/.

[7] Scalog Github Repository. https://github.com/scalog.

[8] SpecLog Github Repository. https://github.com/dasl-uiuc/speclog-artifact.

[9] The Cost of Latency in High-Frequency Trading. https://moallemi.com/ciamac/papers/latency-2009.pdf.

[10] The Milliseconds Market: The Money-Making on High-Frequency Trading. https://fondexx.pro/blog/milliseconds-market-money-making-high-frequency-trading.

[11] The State of Streaming Data Report 2023-24. https://go.redpanda.com/state-of-streaming-data-report-2023-24.

[12] TinyBird: How to Build a Real-Time Fraud Detection System. https://www.tinybird.co/blog-posts/how-to-build-a-real-time-fraud-detection-system.

[13] Apache. Kakfa. http://kafka.apache.org/.

[14] Danial Asif. Building Faster Indexing with Apache Kafka and Elasticsearch. https://careersatdoordash.com/blog/open-source-search-indexing/.

[15] Gourav Singh Bais. How to detect fraudulent clicks in a real-time ad system. https://www.redpanda.com/blog/detect-fraudulent-clicks-real-time-ads.

[16] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[17] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, October 2013.

[18] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR*, volume 11, pages 9–20, 2011.

[19] Shruti Bhat and Kai Waehner. Real-Time Analytics and Monitoring Dashboards with Apache Kafka and Rockset. https://www.confluent.io/blog/analytics-with-apache-kafka-and-rockset/.

[20] Juxhin Dyrmishi Brigjaj. Using Redpanda to build a real-time security IoT platform. https://www.redpanda.com/blog/real-time-security-iot-customer-story.

[21] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-backup Approach. *Distributed systems*, 2, 1993.

[22] Qi Chen, Cheng Liu, and Zhen Xiao. Improving MapReduce Performance Using Smart Speculative Execution Strategy. *IEEE Transactions on Computers*, 63(4):954–967, 2014.

[23] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based Speculative Transaction Execution for Ethereum. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual, October 2021.

[24] Confluent. Motion in Motion: Building an End-to-End Motion Detection and Alerting System with Apache Kafka and ksqlDB. https://www.confluent.io/blog/build-real-time-iot-application-with-apache-kafka-and-ksqldb/.

[25] Confluent. Real-Time Fraud Detection in Banking. https://assets.confluent.io/m/4d949142ef12c4c2/original/20230906-WP-Fraud_Detection.pdf.

[26] Dunith Danushka. Building a scalable IoT data processing architecture with Redpanda. https://www.redpanda.com/blog/streaming-data-platform-for-iot-edge/.

[27] Dunith Danushka. Understanding Event Stream Processing. https://www.redpanda.com/blog/data-streaming-for-financial-services.

[28] Alex Davies. Jump Trading Drives Faster Insights at Scale with Redpanda. https://thenewstack.io/jump-trading-drives-faster-insights-at-scale-with-redpanda/.

[29] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[30] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/-subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[31] Expeed. Get Real-Time IOT Data Analytics Using Apache Kafka And Apache Spark. https://expeed.com/get-real-time-iot-data-analytics-using-apache-kafka-and-apache-spark/.

[32] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. Flexlog: A shared log for stateful serverless computing. pages 195–209, 08 2023.

[33] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[34] Himanshu Gupta. Unlocking a Competitive Edge in Hedge-fund Trading, Right Down to the Data Specifics. https://www.rtinsights.com/unlocking-a-competitive-edge-in-hedge-fund-trading-right-down-to-the-data-specifics/.

[35] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.

[36] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual, October 2021.

[37] Joe Karlsson. Event sourcing with Kafka. https://www.tinybird.co/blog-posts/event-sourcing-with-kafka.

[38] Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. Processing transactions over optimistic atomic broadcast protocols. In *International Symposium on Distributed Computing (DISC 99)*, Bratislava, Slovak Republic, September 1999.

[39] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[40] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[41] Joey Lei. Leveraging Modern Services to Drive Business Agility, Elasticity, and Mobility. https://www.rtinsights.com/modern-services-drive-business-agility-elasticity-mobility/.

[42] Jacob Loveless, Sasha Stoikov, and Rolf Waeber. Online Algorithms in High-Frequency Trading. *Communications of the ACM*, 56(10):50–56, 2013.

[43] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. Lazylog: A new shared log abstraction for low-latency applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 296–312, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[45] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative Execution in a Distributed File System. October 2005.

[46] Artem Oppermann. Detecting fraud in real time using Redpanda and Pinecone. https://www.redpanda.com/blog/fraud-detection-pipeline-redpanda-pinecone.

[47] Filipa Pedrosa and Luís Rodrigues. Reducing the subscription latency in reliable causal publish-subscribe systems. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 203–212, 2021.

[48] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015.

[49] Shyam Purkayastha. Building a real-time data processing pipeline for IoT. https://www.redpanda.com/blog/analyzing-iot-telemetry-data-apache-spark.

[50] RedPanda. RedPanda. https://redpanda.com/.

[51] Redpanda. Understanding event stream processing. https://www.redpanda.com/guides/event-stream-processing-event-sourcing-database.

[52] Dheeraj Remella. How Volt and RedPanda Help Companies Take Real-Time Actions on Streaming Data. https://www.voltactivedata.com/blog/2023/09/volt-redpanda-real-time-actions-on-streaming-data/.

[53] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[54] Rockset. Real-Time Analytics on Kafka. https://rockset.com/sql-on-kafka/.

[55] Raj Sagiraju. Simple, fast, and scalable serverless stream processing with DeltaStream and Redpanda. https://www.redpanda.com/blog/simple-fast-scalable-stream-processing-deltastream.

[56] Salvatore Salamone. Enabling Real-time Analytics At-Scale Use Cases. https://www.rtinsights.com/enabling-real-time-analytics-at-scale-use-cases/.

[57] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 69–84, 2022.

[58] Gang Tao. Case Study: Real-time Fleet Monitoring with Timeplus. https://www.timeplus.com/post/case-study-real-time-fleet-monitoring-with-timeplus.

[59] Gang Tao. Realizing low latency streaming analytics with Timeplus and Redpanda. https://redpanda.com/blog/low-latency-streaming-analytics-timeplus-redpanda.

[60] Rajkumar Venkatasamy. Building a real-time search application with Redpanda and ZincSearch. https://www.redpanda.com/blog/real-time-data-search-redpanda-zincsearch.

[61] VMWare. CorfuDB. https://github.com/CorfuDB/CorfuDB.

[62] Kai Waehner. Fraud Detection with Apache Kafka, KSQL and Apache Flink. https://www.kai-waehner.de/blog/2022/10/25/fraud-detection-with-apache-kafka-ksql-and-apache-flink/.

[63] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vcorfu: a cloud-scale object store on a shared log. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 35–49, USA, 2017. USENIX Association.

[64] Wei Wei, Harry Tian Gao, Fengyuan Xu, and Qun Li. Fast mencius: Mencius with low commit latency. In *2013 Proceedings IEEE INFOCOM*, pages 881–889. IEEE, 2013.

[65] Benjamin Wester, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines Through Client Speculation. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.





Understanding Stragglers in Large Model Training Using What-if Analysis

Jinkun Lin[1], Ziheng Jiang[2], Zuquan Song[2], Sida Zhao[2], Menghan Yu[2], Zhanghan Wang[1], Chenyuan Wang[2], Zuocheng Shi[4], Xiang Shi[3], Wei Jia[2], Zherui Liu[2], Shuguang Wang[2], Haibin Lin[2], Xin Liu[2], Aurojit Panda[1], Jinyang Li[1]

[1]*New York University* [2]*ByteDance Seed* [3]*ByteDance* [4]*Zhejiang University*

Abstract

Large language model (LLM) training is one of the most demanding distributed computations today, often requiring thousands of GPUs with frequent synchronization across machines. Such a workload pattern makes it susceptible to stragglers, where the training can be stalled by few slow workers. At ByteDance we find stragglers are not trivially always caused by hardware failures, but can arise from multiple complex factors. This work aims to present a comprehensive study on the straggler issues in LLM training, using a five-month trace collected from our ByteDance LLM training cluster. The core methodology is what-if analysis that simulates the scenario without any stragglers and contrasts with the actual case. We use this method to study the following questions: (1) how often do stragglers affect training jobs, and what effect do they have on job performance; (2) do stragglers exhibit temporal or spatial patterns; and (3) what are the potential root causes for stragglers?

1 Introduction

Large-language models (LLMs) have been widely adopted and are being used for a variety of tasks. In response to their adoption, many companies are focused on training ever-larger models because empirically an increase in model size has been shown to improve task accuracy. For example, in little more than a year, Meta has moved from the 65 billion parameter Llama-1 model to the 405 billion parameter Llama-3 model. Training these large models in a reasonable time requires using resources, including GPUs and memory, distributed across several thousand servers in a cluster. However, unlike previous distributed parallel data processing workloads such as MapReduce, distributed LLM training requires frequent synchronization and tighter coordination. Consequently, slow workers or *stragglers* can significantly hurt training time and resource efficiency. This paper seeks to answer the question *do stragglers actually pose a serious performance issue in real-world large-scale LLM training deployment?*

The impact of stragglers in LLM training is dictated by the parallelization strategy used to distribute LLM training across a cluster. A typical LLM training job uses a hybrid parallelism strategy that combines multiple parallism strategies, including pipeline parallelism (PP) [12, 19, 29] and data parallelism (DP) [32] to partition state and computation across servers. All of these strategies require frequent coordination to synchronize results across GPUs and servers, and are thus vulnerable to stragglers. A microbatch that is delayed due to a straggler in a PP stage can slow down the entire training batch in DP. Further, few approaches exist to mitigate the effect of stragglers in LLM training: traditional straggler-mitigation approaches [1–3] that rely on backup workers assume infrequent synchronization and would significantly hurt training performance and resource requirements for LLM training jobs. On the other hand, approaches that use asynchronous SGD [9] or drop updates from slow workers [15] change training behavior, and are not widely adopted due to concerns about their effect on model accuracy.

Given that LLM training jobs are vulnerable to stragglers, we set out to understand their effect on real-world LLM training clusters. To do so we analyzed traces collected from the LLM training cluster at ByteDance over a five-month period from January to May 2024 (§3.1). Our analysis seeks to address the following questions:

- How often do stragglers affect training jobs? What effect do they have on training job performance?
- Do stragglers exhibit temporal or spatial patterns? For example, do they appear in most or only a few steps of a training job? Are they confined to a few workers or more widespread across workers?
- What are the potential root causes for stragglers? What is the relative severity of different root causes?

We answer these questions using "what-if" analysis (§3.2): we estimate how a training job would perform if all (or a subset) of stragglers had not occurred. To do so, we identify the *dependency model* within a training job's traced operations, and then perform a simulation based on each operation's non-straggling execution time to quantify job-completion time in the absence of stragglers.

Our analysis revealed that stragglers are widespread and

have notable performance impacts on LLM training; 42.5% of the jobs are at least 10% slower due to stragglers. At the tail, stragglers can result in jobs wasting 45% of their allocated resources (§4.1). Most steps incur similar slowdowns within a straggling job, suggesting that they are often not caused by transient environmental issues but are rather caused by persistent problems (§4.2). Our analysis further shows that stragglers are more commonly caused by a slowdown in computation operations rather than communication operations (§4.3). We did not notice a positive correlation between job size and its straggler-related slowdown, suggesting factors other than job size play a dominant role in straggling (§4.4).

Coupling simulation-based analysis with manual inspection, we investigate several root causes for straggling. We found that hardware or software problems in a server were not a significant cause of the stragglers we observed (§5.1). On the other hand, imbalance in how work was partitioned across pipeline stages (§5.2), an imbalance in sequence lengths included in each microbatch (§5.3), and pauses induced by the garbage collector (§5.4) were responsible for many of the stragglers we observed. We summarize the key observations and their implications in §5.6.

Our analysis pipeline also suggests an approach to make it easier to detect and debug stragglers when a training job is running. As a result, we incorporated portions of our analysis pipeline into a monitoring system called *SMon* (§8) that is now deployed in the training cluster at ByteDance. SMon partially automates the manual analysis we report on in this paper, and is used by the on-call team at ByteDance to detect and address stragglers in important jobs.

2 Background

We start by providing background on distributed LLM training and sources of stragglers.

2.1 Hybrid-Parallel Training and Stragglers

Several different strategies have been developed to parallelize LLM training to overcome memory limitations and reduce training time. Each parallelization strategy is susceptible to stragglers. Below, we introduce the strategies used in our cluster and the sources of stragglers. We describe each strategy in terms of how computation and memory are partitioned across *workers*, a term we use to refer to a single GPU and a process (running on the CPU) that controls it.

Data parallelism (DP), ZeRO [32] and fully sharded data parallelism (FSDP [44]) DP partitions training data across multiple workers, each of which has a replica of the entire model. When DP is used, in each training step, a worker is assigned a training batch. The worker runs forward and backward computation for its assigned batch, and then all workers perform a gradient all-reduce step before moving to the next training step. The gradient all-reduce step requires synchronization between workers, and a slow worker can cause all workers to stall, leading to stragglers. ZeRO [32] and

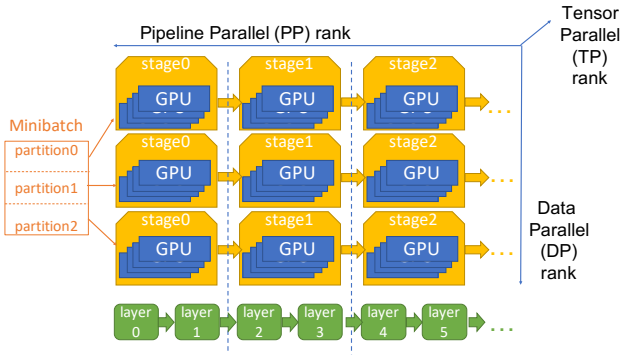


Figure 1: *Rank topology of a DP-PP-TP parallelism.*

FSDP [44] extend DP to reduce the per-GPU memory demand by partitioning the optimizer state, parameters, and/or gradients across workers. Instead of using a gradient all-reduce step, both ZeRO and FSDP require a reduce-scatter step that computes gradients, a device-local parameter update step, and an all-gather step to collect parameters for each training step. The reduce-scatter and all-gather steps require synchronization across workers, and are thus similarly susceptible to stragglers.

Pipeline parallelism (PP). PP partitions the model across multiple workers, each of which holds a disjoint set of consecutive model layers, referred to as a *pipeline stage*. PP reduces the per-GPU memory demand for model weights and activations. During training, a batch of data is split into several microbatches, and training is pipelined through PP stages. Several microbatch scheduling approaches have been proposed, including GPipe [19], 1F1B [12], and virtual pipeline parallelism (VPP) [29]. All of these scheduling approaches assume that computation is evenly partitioned across pipeline stages, and aim to minimize *pipeline bubbles*, *i.e.,* times when a pipeline stage is idle waiting for data from the previous stage. If PP stages are not evenly partitioned, the slowest stage stalls other stages and becomes a performance bottleneck. Thus, PP is susceptible to straggling due to uneven compute partitioning between PP stages. For ease of exposition, we do not explicitly discuss VPP, but our analysis does account for it.

Tensor parallelism (TP) and context parallelism (CP). In addition to PP, one can use Tensor (TP) [23, 29] and context (CP) [11, 26, 35] parallelism to further reduce the per-GPU memory demand. TP partitions each layer's weight between workers, and CP partitions each sequence's tokens across workers. Both require a synchronization step after each transformer layer so that partial results from all workers in the TP or CP group can be aggregated. As a slow device can stall progress during synchronization, both TP and CP are susceptible to stragglers. As we discuss in §7, we do not analyze systematic stragglers in the TP and CP groups.

Hybrid parallelism. In practice, LLM training uses a hybrid strategy that combines all of the parallelism strategies discussed above and provides better performance than any in-

dividual strategy.[1] When hybrid parallelism is used, workers can be organized into a hypercube, where each dimension corresponds to one parallelism strategy. A worker's coordinate provides its *rank* (*e.g.,* pipeline stage) in each parallelism dimension. Workers are also assigned a unique *global-rank* for identification. Figure 1 shows an example worker topology with DP-PP-TP parallel training.

Training with hybrid parallelism operates in a layered fashion. For example, when DP-PP-TP parallelism is used, workers with the same DP rank are grouped into a single PP group, which is responsible for computing gradients for a training batch. Inside each PP group, workers with the same PP rank are grouped into a TP group, each of which is responsible for a single PP stage. Finally, workers within the same TP group use TP parallelism to compute each layer in the PP stage.

The case of straggling for hybrid-parallelism. Hybrid-parallelism is affected by stragglers appearing in any of the parallelism strategies that it combines. For example, in the DP-PP-TP parallelism strategy, a slow TP worker will slow down the PP rank to which it belongs, leading to a PP bubble. This PP bubble, in turn, will slow down the entire DP rank to which the worker belongs, delaying gradient synchronization and stalling other DP ranks.

2.2 Goals and Challenges

Straggler definition. An LLM training job that uses hybrid parallelism is straggler-free if all workers take the same amount of time to complete their assigned work. This minimizes the time required for synchronization, resulting in an ideal straggler-free scenario that achieves the best possible performance. We also note that, by this fairly broad definition of straggling, a worker lagging behind others for any reason is considered a straggler. These reasons can include not only hardware issues that deterministically affect one single worker, but also unpredictable stalls that affect all workers uniformly (*e.g.,* garbage collection that prevents a task from making progress [22, 38]), as well as workload imbalance due to data skew [31, 43] or poor workload partitioning across weights (*e.g.,* PP stage) [11].

Our goal. In this paper, we want to investigate how straggling impacts real-world LLM training jobs and to unearth some of the underlying root causes. To do so, we seek to quantify *how much an actual LLM training job's speed differs from that of some ideal straggler-free case*. In keeping with our earlier definition, a straggling worker is one that completes its work more slowly than others. Thus, in the presence of stragglers, a job is delayed by its need to synchronize across groups of workers, resulting in a slower speed compared to the ideal straggler-free scenario.

Our study is based on analyzing traces from LLM training jobs collected over a five-month period. Although these traces capture each job's actual execution time, estimating the corresponding straggler-free completion time remains challenging. This is because it is tricky to assess straggling operations' impact on the overall job duration in the presence of overlapping execution. Traditional critical path analysis [5] falls short in this context, as highly parallel and homogeneous workloads like LLM training can exhibit many similarly critical paths. Focusing on a single path can lead to misleading conclusions, as shown in Coz [8]. To address this, we instead use trace-based simulation to "execute" each job on an alternative timeline where straggling operations are brought in line with their peers.

[1] Other parallelisms such as expert parallelism (EP) [24] also exist but are not used in the jobs that we analyze.

3 Methodology

In this section, we describe the traces used for our study and our methodology for analyzing them.

3.1 The LLM Training Job Traces

The cluster setup. The cluster on which the traces have been collected is dedicated for training and shared internally by multiple teams. The machines used in the cluster have a similar hardware configuration as NVIDIA's DGX servers: each server is equipped with eight GPUs interconnected with NVLink or PCIe links, four or eight NICs with hundreds of Gbps bandwidth, a separate NIC used for storage and management, a few hundred CPU cores and a few TBs of memory. Servers are interconnected through high-performance switches configured in a three-layer CLOS topology. The network is overprovisioned and carefully tuned [22] to ensure that there are no slowdowns due to network congestion.

Multiple jobs can be scheduled to run on the cluster at the same time, with each job having exclusive use of its allocated GPUs for the duration of the job. The scheduler ensures that each job uses homogeneous hardware by allocating it the same type of GPUs. Furthermore, the scheduler performs the GPU allocation in a best-effort manner to ensure a job's GPUs are close together in the network topology. Since large jobs request GPUs in multiples of eight, different large jobs do not share the same server machine. Combined with the lack of network congestion, this means we do not see stragglers due to resource contention despite jobs running on the same cluster.

Trace collection. The traces used in our study are collected from LLM pretraining jobs submitted during the time period from 2024/01/01 to 2024/05/31. As our study focuses on large jobs, we only use traces for jobs that use at least 128 GPUs. We also discard traces that are invalid or not suitable for analysis as detailed in §7. This produces 3079 jobs for our analysis. These jobs contain both dense and mixture-of-experts (MoE) models that are configured to train using either short or long context sequences. Of the 3079 jobs, 31.7% use $\geq$ 256 GPUs, 18.3% use $\geq$ 512 GPUs, and 3.6% use $\geq$ 5000 GPUs. In total, the analyzed jobs cover around half of the total GPU-hours among all LLM training jobs.

	Type of Operation	Description
Compute Op.	forward-compute	Forward computation done for a microbatch for a PP stage.
	backward-compute	Backward propagation done for a microbatch for a PP stage.
PP-specific communication Op.	forward-send, forward-recv	P2P communication between PP stages for a microbatch on the forward pass.
	backward-send, backward-recv	P2P communication between PP stages for a microbatch on the backward pass.
DP-specific communication Op.	params-sync	Collective communication (all-gather) among all DP ranks for a given PP stage to retrieve the stage's weights prior to the first microbatch's forward-compute.
	grads-sync	Collective communication (reduce-scatter) among all DB ranks for a given PP stage to aggregate the stage's gradients after the last microbatch's backward-compute.

Table 1: *Types of operations traced by NDTimeline and their description.*

All jobs in our traces are done using customized versions [22] of the open-source distributed LLM training system, Megatron-LM [28]. The jobs are configured to use different combinations of the parallelization strategies supported by Megatron-LM including DP, PP, TP and CP. [2]

The Megatron-LM based training system has been instrumented with our in-house profiling tool *NDTimeline* [20]. By default, it samples 10% of the training steps (aka iterations) of a job for profiling. For each profiled step, the tool records the start and end time of a set of operations that are deemed important, including those performing forward and backward computation as well as those performing communication. Table 1 gives all the profiled *operation types* and their description. A profiled forward or backward compute operation consists of many GPU kernels in order to reduce the cost of tracking many small computation kernels. NDTimeline periodically synchronizes the clocks of all machines for a job, thereby allowing us to align related operations across different machines for the purpose of what-if analysis (§3.2).

For each operation in the trace, its logged entry contains the operation type, its start and end timestamps as well as a set of metadata including the training step ID, the microbatch ID, the PP rank (within each PP group) and the DP rank (within the overall DP group). These metadata enable us to reconstruct the operation dependencies which are necessary in order to simulate an alternative timeline had stragglers not been present (§3.2).

3.2 Simulator for What-if Analysis

The goal of the what-if analysis is to assess the impact of stragglers by answering the following questions:

1. *how long would a job take if all stragglers were absent?*
2. *how long would a job take if all except for a certain group of stragglers were absent?*

To answer these questions, we simulate an alternative timeline without stragglers. The main insight is that comparable operations would have the same duration in the absence of stragglers. Based on this insight, we first try to estimate the duration of each operation in the alternative straggler-free scenario. Then, we simulate an alternative timeline in which operations launch according to their dependencies and complete within their estimated idealized durations. By comparing the simulated job completion time (JCT) to that of the actual trace, we can assess the impact of stragglers.

Estimate idealized operation durations for straggler-free scenarios. Conceptually, we can organize the traced operations into a multi-dimensional tensor with four dimensions: training step, microbatch, PP rank and DP rank. Let's refer to this tensor as an *OpDuration* tensor. We have one such tensor for each operation type in Table 1. For compute operations, the elements in the tensor are simply the corresponding operation duration in the trace. For communication, we calculate a sub-part of the traced duration called *transfer-duration*. As communication is done as part of a collective (or a P2P pair), the traced duration of an individual operation is affected by two factors: 1) the amount of time to transfer the data to another rank, aka the "transfer-duration". 2) the amount of time to wait for the other operations in the collective (or P2P pair) to start, aka the "blocking-duration". Of the two factors, "transfer-duration" is intrinsic to the collective operation while the "blocking-duration" is determined by operation scheduling. Therefore, we only store the "transfer-durations" in the *OpDuration* for a communication operation type, e.g. *params-sync*, *forward-send* etc. To estimate an operation's "transfer-duration", we take the maximum start time among all of its peer operations in the same collective (or the same P2P pair) and subtract it from its end time.

All operations of the same type handle the same workload, implying that, in the absence of stragglers, all elements of the idealized *OpDuration* tensor would be equal. To assess the impact of a particular group of stragglers—such as those specific to a machine with PP-rank p and DP-rank d—we selectively "fix" straggling on all other machines by overriding the duration of operations on those machines with their idealized values. For those elements belonging to the straggling machine (aka OpDuration[:,:,p,d]), we keep their original durations. We adopt a similar approach (in §5) to estimate the effect of only fixing some straggling elements: we use idealized operation times only for the elements that need to be fixed, and leave the others unchanged.

[2] CP in our traces uses an in-house implementation with a slightly different algorithm than the CP in open-source Megatron-LM.

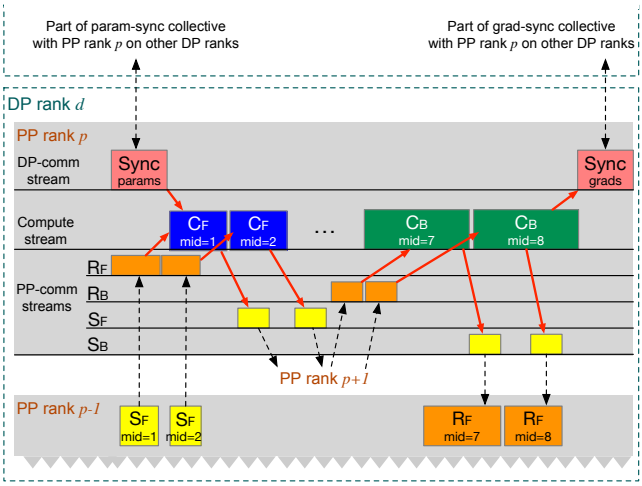


Figure 2: *The dependency model used in simulation. "mid" denotes microbatch ID. Operations within a stream execute sequentially. Across streams on the same rank, receive operations (e.g., R_F, $Sync_{params}$) precede the corresponding compute operation requiring the data. The opposite dependencies exist for compute and send operations. Across DP ranks, params-sync ($Sync_{params}$) and grads-sync ($Sync_{grads}$) on the same PP rank form a collective group. Across two adjacent PP ranks, their send and receive operations form a P2P pair.*

Once we determine the parts of *OpDuration* tensor that should have the same idealized values in a hypothetical straggler-free setup, the question remains what that value should be. We use different methods for computation and communication operations. For the computation operation, we use the average value across the group of elements to be equalized. For communication operations, we use median instead of average. Initially, we used average for both types of operations, but changed our solution after some manual root cause investigation. In particular, we observe that computational straggling arises predominantly due to workload partitioning imbalance. Therefore, taking the average makes sense as doing so amounts to workload re-balancing. On the other hand, communication operations have identical transfer volumes across different training steps, microbatches, PP ranks and DP ranks, but they straggle due to external issues like switch/NIC flapping. Furthermore, the affected operations tend to be very long, which significantly skews the average value and makes median a better choice.

Extract operation dependencies. Our simulator requires two pieces of input: the idealized operations durations and the operation dependency model. We have previously discussed how to estimate idealized operation duration. Next, we will explain the simulator's dependency model, which is derived from the Megatron-LM based training system in use.

In this dependency model, each worker runs several "streams" to execute its operations. All operations scheduled to run on a single stream execute sequentially, while operations across streams execute concurrently as long as their dependencies are satisfied. More concretely, each worker has a stream executing all forward and backward compute operations, a stream executing all DP-specific communication operations and four streams each executing a different type of PP-specific communication operations: forward-recv (R_F), backward-recv (R_B), forward-send (S_F) and backward-send (S_B), as shown in Figure 2. The dependencies among operations are as follows:

- *Same stream dependency:* Operations inside a stream are sorted according to their launch time in the trace. We assume an implicit dependency between adjacent operations since operations on the same stream execute sequentially.
- *DP communication and compute dependency:* The first microbatch's forward-compute operation at each PP stage should happen after the corresponding params-sync collective communication to fetch that stage's parameters, as shown in Figure 2 ($Sync_{params} \rightarrow C_{F,mid=1}$). The parameters are cached locally and used for subsequent microbatches. The gradients computed for different microbatches are locally accumulated and then aggregated across DP ranks. Thus, the last microbatch's backward-compute should happen before the grads-sync collective communication to aggregate the gradients of that PP stage across DP ranks, as shown in Figure 2 ($C_{B,mid=8} \rightarrow Sync_{grads}$).
- *PP communication and compute dependency:* Except for the first PP rank, the forward- and backward-compute operation of a microbatch on PP rank p must start after the completion of that microbatch's forward- and backward-receive communication operation on the same GPU, as shown in Figure 2 (e.g. $R_{F,mid=1} \rightarrow C_{F,mid=1}$, $R_{B,mid=7} \rightarrow C_{B,mid=7}$). Similarly, except for the last PP rank, the forward- and backward-send operation of a microbatch on PP rank p must start after the completion of that microbatch's forward- and backward-compute operation on the same GPU, as shown in Figure 2 (e.g. $C_{F,mid=1} \rightarrow S_{F,mid=1}$, $C_{B,mid=7} \rightarrow S_{B,mid=7}$).
- *Cross-rank communication dependencies:* The DP communication operations (aka params-sync, grads-sync) for a given microbatch form a collective group among all DP ranks with the same PP rank. Similarly, the PP send and receive operations for a given microbatch form a pair between adjacent PP ranks with the same DP rank. The dependency model for a group of collective (or P2P) operations is that no individual operation can start its data transfer until all operations have been launched.

Simulate an alternative timeline. Using the dependency model and idealized durations, we can simulate an alternative execution timeline using the following rules:

- The simulator launches an operation as soon as all of its dependent operations finish. In other words, an operation's start time is calculated as the maximum end time of its dependent operations.
- A computation operation after launch is immediately

marked as finished with end time calculated as the start time plus the corresponding operation duration in *OpDuration*.

- For each communication operation, the simulator waits for all its peer operations in the same collective group (or P2P pair) to launch. An operation's end time is calculated as the maximum launch time among the group plus the corresponding transfer-duration stored in *OpDuration*.

3.3 Metrics for Straggler-related Slowdown and GPU Waste

Now that we have designed a simulator to estimate a job's completion time in alternative straggler-free timelines, what metrics should we calculate to quantify the impact of stragglers? Let us use T_{ideal} to denote the JCT without stragglers. Similar to [31], to account for errors introduced from simulation, we also simulate the original timeline using unmodified operation durations and denote the resulting JCT as T. The simulation errors are relatively small and reported in §6. Furthermore, a small number of our traces had large simulation errors, and to ensure analysis fidelity, we discard traces with simulation error $\geq 5\%$ (§6). To quantify straggler-related performance slowdown, we calculate the slowdown metric S as a ratio:

$$S \triangleq T/T_{ideal}. \tag{1}$$

In addition to the overall straggler-related slowdown, we also want to quantify slowdowns due to straggling in different types of operations (Table 1). To do so, we first compute the *operation-type slowdown* S_t for each operation-type t as

$$S_t \triangleq T_{ideal}^{-t}/T_{ideal}. \tag{2}$$

where T_{ideal} is the ideal JCT computed as described above, and T_{ideal}^{-t} is the JCT when elements in *OpDuration* for operation-type t are not fixed.

In our cluster, a job has exclusive access to a GPU for its entire duration. As a result, an increase in job completion time (or a slowdown) can be directly translated to the amount of GPU-hours wasted by the job. Concretely, we estimate a job's *resource waste* as the percentage of GPU-hours wasted using the equation:

$$\frac{T - T_{ideal}}{T} = 1 - \frac{1}{S}. \tag{3}$$

Similarly, we can calculate wasted resources due to different operations types by computing $1 - 1/S_t$.

4 Impact of Stragglers

We perform simulation-based what-if analysis on the job traces and report the impacts of stragglers on our cluster.

4.1 Stragglers are prevalent and cause non-negligible resource waste.

Figure 3 plots the cumulative distribution function (CDF) of the resource waste percentage among all jobs. We observe that

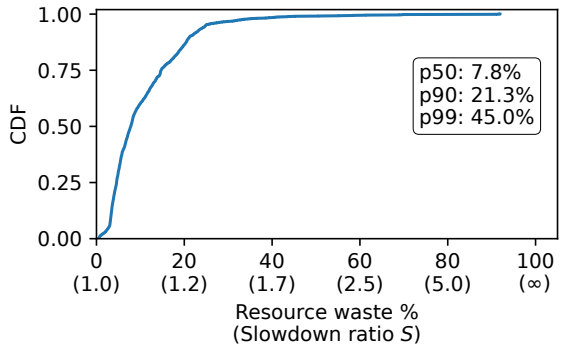


Figure 3: *CDF of resource waste among all jobs. A given waste percentage's corresponding slowdown ratio is shown in parentheses.*

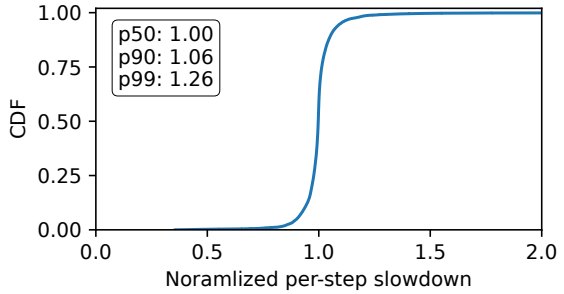


Figure 4: *CDF of per-step slowdowns normalized by job slowdown for 15 randomly selected steps from each straggling job.*

42.5% of the training jobs that we trace straggle. Furthermore, because of stragglers, $> 10\%$ of the jobs waste at least 21.3% of the GPU-hours it is allocated, while $\sim 1\%$ jobs waste at least 45.0% of its allocated GPUs. Across all our traces we found that 10.4% of the allocated GPU hours are wasted due to stragglers.

We also investigated jobs with large slowdowns ($S > 3$), and found that all of them were large jobs, where fewer than 3% of the workers were responsible for the problem. Furthermore, in most cases, the slow operations were performing computation, rather than communication. From this, we hypothesize that server problems (which could be problems with the hardware or misconfiguration) are generally to blame for these problems.

4.2 Steps have similar slowdowns in a straggling job.

Next, we investigate whether a few really slow steps or most steps contribute to a straggling job's slowdown. We refer to a job as straggling if it has a significant slowdown ratio $S > 1.1$. To do so, we define a step's slowdown as the ratio of the step's execution time to the ideal step execution time (*i.e.*, $\frac{T_{ideal}}{n}$ for a job with n training steps). Figure 4 shows the CDF of per-step slowdown normalized by the job's overall slowdown ratio. We observe that the median step has a normalized slowdown of 1.0, and even the 90th percentile step has a normalized slowdown of only 1.06. This indicates that most steps have a slowdown similar to the job's overall slowdown, suggesting

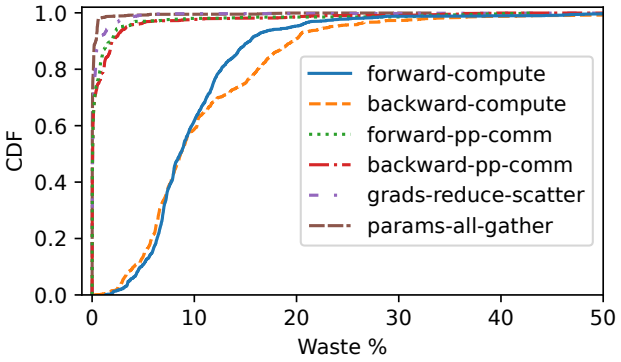


Figure 5: *Resource wasted due to the slowdown of a single type of operation. Computation slowdown caused the most resource waste, while communication had a minimal impact.*

that stragglers are not caused by temporary environmental factors but by more persistent problems. This result also implies that it is sufficient and cost-effective to sample a few training steps to profile stragglers.

4.3 Which operation types are to blame for straggling?

Figure 5 shows the fraction of resource waste by each operation type across all jobs in our traces. In our traces, communication time measures time spent transferring data, and thus a slowdown in send times produces a corresponding slowdown in receive times. As a result, we group both together in the figure. Contrary to the observation in FALCON [41] we find that most slowdowns are caused by computation operations, as opposed to communication. This is owing to our ample network bandwidth, the use of dedicated LLM-training clusters, and several in-house network optimizations and tunings [22].

Figure 5 also shows that PP-level communication has a slightly higher impact than the DP-level communication operations. This matches our expectation since DP-level communication is overlapped by a lot and can tolerate more slowdowns than PP-level communication kernels, many of which happen in the warmup and cooldown phases, which are on the critical path of a training step.

4.4 How does job size correlate with straggling?

Surprisingly, we do not observe an obvious positive correlation between the slowdown and job size. This suggests that job size is not the determining factor of straggling, and other factors like the model type or human factors may play a more important role. For instance, very large jobs are usually babysat by our on-call team and optimized better than others. Therefore, their slowdowns are not necessarily worse than those for smaller jobs. As another example, long-context jobs are observed to suffer more from stragglers (§5.3), but they typically have smaller job sizes, which when mixed with other models biased the result towards the opposite job size and slowdown correlation.

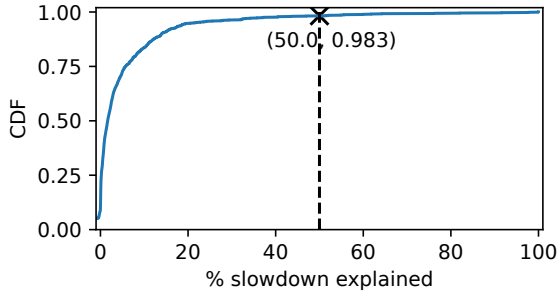


Figure 6: *CDF for M_W, the performance recovered after fixing the slowest 3% of workers, normalized by the performance recovered after fixing all workers. Observe that the slowest 3% of workers do not contribute significantly to the observed slowdown for most jobs.*

5 Root Causes

In this section, we analyze the traces to diagnose the root causes of stragglers. Our analysis focuses on straggling jobs, aka those with slowdown $S \geq 1.1$. Identifying the root cause of a job slowdown requires manual examination, and thus we cannot investigate all possible root causes. Instead, we focus on common root causes, i.e., ones that are either likely to cause stragglers or are commonly to blame in our setting. We start by looking at whether faulty hardware (or software) in one (or a few) workers is to blame (§5.1) Next, we discuss three common root causes in our cluster: imbalance in how computation is partitioned across pipeline stages (§5.2), imbalance in the length of sequences in each microbatch (§5.3), and stalls during garbage collection (§5.4). Finally, in §5.5, we briefly describe two interesting, but uncommon causes of stragglers.

5.1 Are individual workers to blame?

We first analyze how many of the stragglers are due to hardware or software problems with workers. Because we run health-checks, we expect that only a small fraction of workers should exhibit any problems. Consequently, if a job is slow because of problems with a few workers then fixing the execution time of operations running on the problematic workers (by changing them to the ideal execution time) should suffice to fix the entire job's completion time.[3] We use this observation to measure the impact of problematic workers on straggling jobs.

We start by computing the slowdown attributed to each worker, using the same technique that we used in the previous section to estimate an operation's slowdown percent. Specifically, we define the slowdown S_w for worker w as

$$S_w \triangleq T_{ideal}^{-w} / T_{ideal}. \quad (4)$$

where T_{ideal}^{-w} is the job completion time (obtained from the

[3]Though one faulty machine with a network issue affecting DP communication will affect additional $DP_degree - 1$ workers participating in the same DP-communication, as shown in §4.3 slowdown of DP communication is not a major contributor to straggling jobs, and consequently we do not consider it.

simulator) when we only fix operations that were not executed on worker w, and T_{ideal} is the execution time when all operations are fixed.

Next, for each job, we select the set of workers W whose slowdowns S_w are the highest 3% of the job. If a small number of workers are problematic due to hardware (or software) problems, then W would contain those workers. We then compute the fraction of the job's slowdown that can be attributed to these workers, M_W

$$M_W = \frac{T - T^W_{ideal}}{T - T_{ideal}}, \tag{5}$$

where T is the simulated original step duration (without any stragglers being fixed), T^W_{ideal} is the simulated step duration when only operations running on the selected workers are fixed, and T_{ideal} is the ideal simulated step duration after all stragglers are fixed.

Computing M_W for large jobs (with thousands of workers) is expensive because it requires running thousands of simulations to compute S_w for each worker w. Therefore, we use an approximation to scale our analysis: rather than computing a slowdown for individual workers, we measure a slowdown for whole DP ranks and PP ranks. We then assign each worker the smaller (*i.e.,* minimum) of the slowdowns for the DP and PP ranks to which it belongs (as a reminder, any worker must belong to one DP rank and one PP rank). This allows us to reduce the number of simulations from *DP degree* $\times$ *PP degree* to *DP degree* $+$ *PP degree*, making computation feasible.

Figure 6 shows a CDF of M_W for straggling jobs. We observe that worker problems contribute to more than 50% of the observed slowdown for only 1.7% of straggling jobs. This leads us to conclude that problematic workers are not the dominant factor for a majority of the straggling jobs in our traces.

We further investigated the few cases where problematic workers were primarily responsible for job slowdowns, and found that they resulted in a much larger slowdown: slowdown for jobs with problematic workers is 3.04 compared to the average slowdown of 1.28.

5.2 Stage Partitioning Imbalance

In our traces, we observed that an imbalance between the computation performed by the last pipeline stage and other pipeline stages was a common cause of stragglers. The last pipeline stage needs to execute a loss layer, and in most cases, the loss layer requires more compute cycles than transformer layers (and embedding layers, which take negligible compute time) [11, 39]. Thus, it is easy for users to accidentally partition the model (e.g., by evenly dividing layers over pipeline stages) in a way that leads to the last stage requiring significantly more computation, which in turn leads to stragglers.

To verify that the loss layer took longer than the transformer layers, we ran a job with four pipeline stages, where each pipeline stage runs 9 transformer layers, and the last stage runs an additional loss layer. The PyTorch trace for this job shows that logit computation (in the last layer) is over 9 times longer than a transformer layer, and that as a result, forward-compute (backward-compute) of the last stage is 2.07 (1.41) times slower than an average stage's computation.

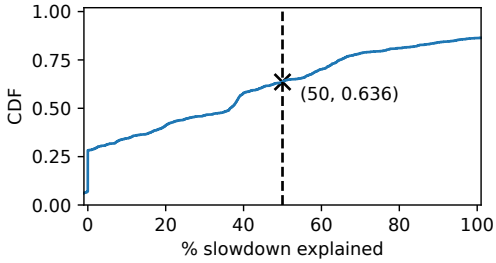


Figure 7: *CDF for M_S, the performance recovered after fixing all workers on the last PP stage, normalized by the performance recovered after fixing all workers. We set $M_S = 0$ for 21.1% of jobs not running PP. Observe that the last stage slowdown is a common straggler issue.*

We use a similar approach as in §5.1 to study the prevalence of this problem: we compute in simulation $T^{lastStage}_{ideal}$, the job's completion time if we only fix the execution time for operations in the last pipeline stage. Next, for each job that uses pipeline parallelism, we compute M_S, the contribution of the last stage as $M_S = (T - T^{lastStage}_{ideal})/(T - T_{ideal})$ (and we set $M_S = 0$ for the 21.1% of jobs that do not use pipeline parallelism).

Figure 7 shows the CDF of M_S's across jobs. We observe that for 39.3% of jobs, a majority of the job slowdown (*i.e.,* $M_S \geq 0.5$) is because of the last pipeline stage.

Next, we looked at whether this problem could be mitigated. We already use an approach similar to Llama 3 [11] to address this problem: we assign ε fewer layers to the last pipeline stage. However, we need to manually tune ε, and this has proven to be challenging for several reasons: First, when partitioning the model, we must assign an entire transformer layer to a pipeline stage, which limits how computation is split between pipeline stages; Second, the time taken by the loss layer (in proportion to the transformer layer) increases either if the vocabulary size grows, or if the maximum-sequence-length or the hidden layer size decreases. Larger ratios are likely to become more common as vocabulary size increases [34], and consequently, as others have also observed concurrently [42], this is going to increase the number of transformer layers that need to be executed in earlier pipeline stages to ensure that they take the same time as last stage. This in turn limits the number of pipeline stages that can be used for these models and leads to more layers in each virtual stage than desirable, limiting model parallelism. Consequently, even a good value of ε can result in suboptimal performance.

We also attempted to manually tune the number of layers in each pipeline stage for the job (with 9 transformer layers) that we discussed above, and we found that with manual partitioning we could get a speedup of 9.9%. However, even with

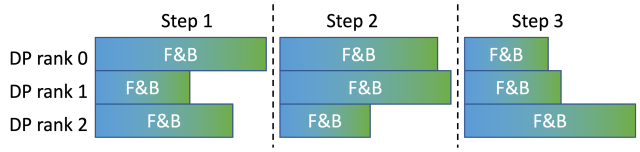


Figure 8: *A representative timeline for jobs with sequence length variance and using pure data-parallelism. Large variance exists in the computation operation times, causing stragglers to happen on some random DP rank(s) every step. "F&B" blocks denote the duration from when the first forward-compute launches to when the last backward-compute finishes.*

manual tuning, the computational load is not perfectly even across stages, *e.g.,* the forward-compute of the last stage is 1.55X of the other stages after tuning.

5.3 Sequence Length Imbalance

Next, we analyzed jobs whose slowdown is not explained by slow workers (§5.1) or imbalance in the computational load between pipeline stages (§5.2). Our inspection revealed that for long-context jobs, differences in sequence lengths between training data are a significant contributor to job slowdown. Recent studies [4, 14, 21] have shown that the datasets used to train long-context LLMs have a long-tailed sequence length distribution. We confirmed that this was also the case in our cluster: Figure 10 shows the sequence length distribution for the training data used in a long-context training job, where the *maximum-sequence-length* is set to 32K.

Variance in sequence length is a problem because the algorithmic complexity of the self-attention layer is quadratic [23]. Our system forms a training microbatch by collecting sequences (chosen at random) until the total length of the microbatch reaches a predefined maximum-sequence-length. However, the computation time for the microbatch depends not just on the maximum-sequence-length, but also on the length of each sequence s_i that is included in the microbatch because the computation time for a microbatch is $O(\sum s_i^2)$. For instance, a microbatch with one sequence of length 32K requires 32× more compute time than a microbatch containing 32 sequences of length 1K. The difference in computation time across microbatches results in bubbles in pipeline stages, and in DP ranks finishing at different times (shown in Figure 8). Both combine to produce stragglers.

Our reasoning above is based on the observation that computation time is proportional to $O(\sum s_i^2)$. In Figure 9, we empirically verify this assumption: we plot the duration and $\sum_i s_i^2$ of each microbatch in the first several training steps of a representative job, and find that indeed they are proportional.

Next, we quantified the fraction of jobs whose slowdown is due to sequence length imbalance. We cannot use the same metric as the previous two sections because our trace does not provide us with enough information to correct for sequence length imbalance. Instead, we measure the occurrence of slowdown due to sequence length imbalance using a *forward-backward correlation* metric. Our metric is based on the ob-

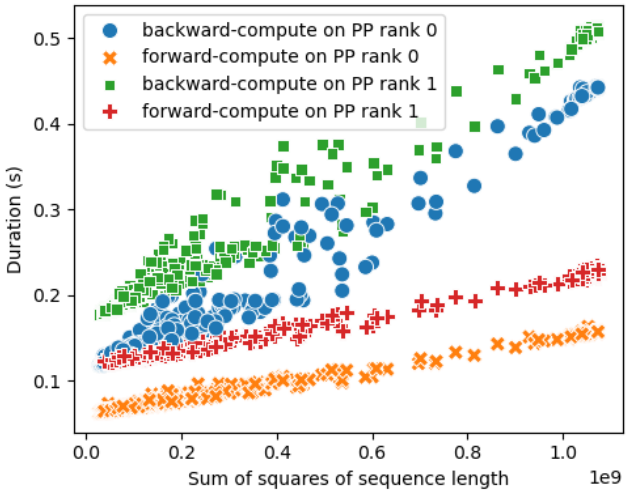


Figure 9: *Microbatch computation duration vs sum of sequence length squares, profiled over dozens of training steps on a job with maximum-sequence-length of 32K. Each data point represents a microbatch on the forward or backward pass on a specific PP rank.*

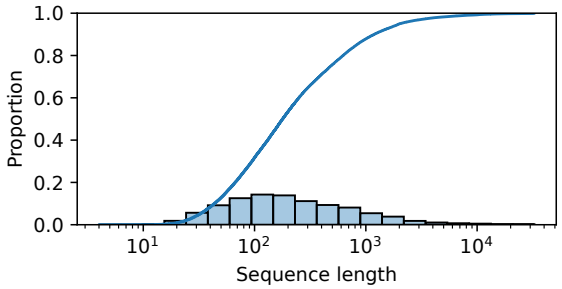


Figure 10: *Sequence length distribution, collected on a job with a maximum-sequence-length of 32K. The histogram is shown in bars, and the CDF is shown on the curve. Note the log scale on the x-axis.*

servation that if forward-compute for a microbatch is slow because of sequence length imbalance, then the backward-compute must also slow down by a similar amount, and the forward- and backward-compute times must be correlated. Figure 11 shows the Pearson correlation factor for a pipeline stage[4] in each straggling job (i.e., those with $S \geq 1.1$). Empirically, we found that jobs with a correlation coefficient ≥ 0.9 were most likely to have been slowed down because of sequence length imbalance. Using this threshold, we observe that 21.4% jobs are affected by sequence length imbalance. They have an average slowdown of 1.34.

Furthermore, in Figure 12 we analyzed how a change in the maximum-sequence-length affects job slowdown. We observed that sequence length imbalance has a larger effect as the maximum-sequence-length grows. But context lengths are growing, and thus addressing this scalability challenge is

[4]To avoid noise brought by microbatches computing loss and embedding layers, we only use microbatches on the second PP stage when *PP degree* ≥ 3. Otherwise, we use the first stage, and if VPP is used we also drop the first virtual stage to discard microbatches containing embedding layers.

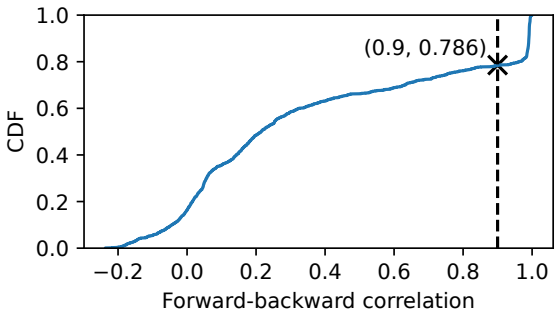


Figure 11: *CDF of forward-backward correlation of all straggling jobs in the solid line. The dashed line shows the proportion of overall resource waste by the jobs with correlation under the x values.*

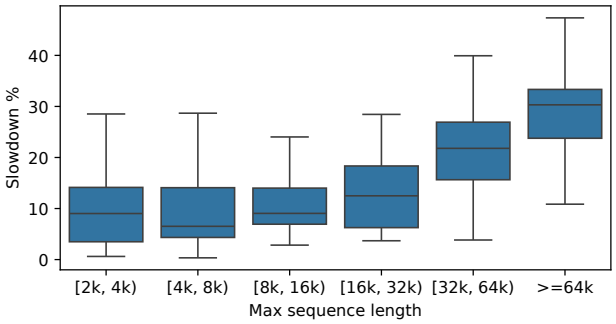


Figure 12: *Long context vs others.*

becoming increasingly important.

One approach to addressing this problem is to distribute sequences across microbatches so as to equalize the computation time for all microbatches. As we discussed above, microbatch computation times can be accurately predicted given sequence lengths (Figure 9). Thus, we prototyped a version of this approach where after a batch is formed, we use a linear model to redistribute sequences so that all DP ranks have a balanced computational load. We formulated the redistribution problem as a multiway number partitioning problem [40] and solved it using a greedy algorithm similar to the one used by DistTrain [43].[5] We used PyTorch's distributed KVStore [37] to exchange sequences between DP ranks. Next, each DP rank divides the assigned sequences into microbatches so that the sums of sequence lengths across microbatches are balanced, and we achieve this with a greedy algorithm. We tested this approach on a representative job with a maximum-sequence-length of 32K, and observed a 23.9% improvement in throughput.

Deploying such a fix in practice requires evaluating its effect at scale, and we leave this to future work. We expect that our proposed fix might increase memory requirements, because balancing computation in this manner results in sequence length sums varying across DP ranks, and might lead to increased memory requirements for some ranks. Furthermore, this fix only solves the imbalance at the DP level, but similar to DistTrain and DynaPipe [21], we also observed imbalance at the PP level (and a large number of PP bubbles) in jobs with large PP degrees and more microbatches. Other approaches to balancing computation across microbatches might be required to address PP-level imbalance.

[5] Except that we sort sequences in descending order since it gives a much better result in our case.

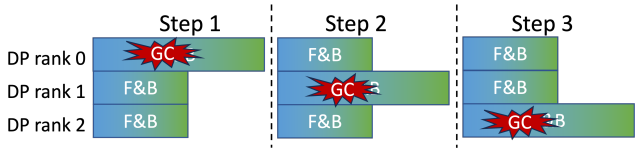


Figure 13: *Representative timeline for jobs suffering from the GC straggler. Different workers perform their GC at different steps, straggling each other.*

5.4 Python's Automatic Garbage Collection

Garbage collection (GC) is another significant cause of slowdowns. The jobs in our cluster are run using Python, whose runtime triggers GC when it is deemed necessary [36]. Once triggered, GC can take 100s of milliseconds, and during this period the user program is paused and new kernels cannot be started,[6] stalling forward-compute operations. Note that backward-compute operations are not affected because they are launched from C++.

This problem is further exacerbated because different Python processes trigger GC at different times. While the GC pauses a single worker, it blocks the entire training job as shown in Figure 13. The number of such pauses grows as the number of workers increases, and thus stragglers due to GC have become a more important concern as models have increased in scale.

Therefore, in 2023 our engineering team implemented a planned GC optimization to mitigate stragglers due to GCs. This optimization turns off Python's automatic GC mechanism, and instead manually schedules GC at a user-specified interval (that is specified in terms of training steps). This ensures that all workers run GC at the same time. This optimization is effective at preventing GC-related stragglers: on a job that uses 128 DP ranks, using this optimization so that GC is run every 500 training steps results in a 12.6% improvement.

While the planned GC optimization is effective, it is challenging to use in practice because choosing an appropriate GC-interval is hard. Picking too large a GC-interval can lead to the job running out of memory and crashing, while choosing too small a GC-interval can lead to worse performance. But the rate at which memory is allocated, and thus the appropriate GC-interval varies by job, and thus users need to tune the interval for each job. Therefore, we are conservative and do not enable planned GC by default: users can turn it on after they have analyzed previous runs of a job to determine an appropriate GC-interval. Consequently, this fix has not been widely adopted in our cluster.

[6] Python uses a stop-the-world garbage collector.

We also observed that the amount of time for which the GC pauses execution increases as the job progresses. This leads to a gradual decrease in training throughput, as has also been observed in the past [22, 38]. We suspect the slowdown is due to a memory leak, which increases heap size as the job runs longer and leads to longer GC pauses. Interestingly, we found that planned GC can mask the impact of this leak and leads to sustained training throughput.

5.5 Other Root Causes

As we stated previously, the set of root causes we have discussed is not exhaustive, and jobs in our traces might have been slowed down by other problems. Rather, our goal was to analyze some of the more obvious or common causes of stragglers. Below, we describe two uncommon but interesting root causes that appeared in our traces.

CUDA memory fragmentation. In a few cases, we observed that memory fragmentation slowed down PyTorch's CUDA memory allocator, and led to a significant increase in calls to `cudaFree` and `cudaMalloc`, which led to slower than usual forward- or backward-compute operations. In our traces, this appears as cases where TP communication kernels running on different TP ranks in the same TP group start at different times but finish roughly at the same time, suggesting that some of the communication kernels experience launch delays. We examined the PyTorch traces for these jobs to confirm that they did indeed make several `cudaFree` and `cudaMalloc` calls, and then we enabled allocator memory tracing and reran the job to further understand the problem. During rerun we observed many `segment_alloc` and `segment_free` calls, confirming that the PyTorch memory allocator was to blame. In future work, we plan to study both the prevalence of this problem and mitigation strategies.

False kernel dependency. When we first started training large MoE models, we found that reduce-scatter kernels (used to synchronize gradients) block other kernels (that do not depend on the reduce-scatter kernel) from being launched, and lead to significant job slowdown. We suspect that this is because of false dependencies [13, 30] caused by unrelated kernels sharing the same CUDA hardware queue. We found that increasing `CUDA_DEVICE_MAX_CONNECTIONS` alleviates the problem. However, we have also found that the problem seems to occur (and then disappear) as we evolve our model and framework, and we continue to investigate it to identify the cause.

5.6 Summary: Observations and Implications

Our root cause analysis yields the following key observations about our traces:

- Relatively few stragglers were due to machine problems (either due to hardware or software), implying that traditional health metrics are unlikely to aid in detecting or preventing most stragglers.
- The most prevalent causes for straggling are due to uneven pipeline stage partitioning, imbalance in sequence lengths in each microbatch, and pauses induced by the garbage collector. Our observations also led us to develop a new approach to address sequence length imbalance (§5.3).
- We observed two new but less prevalent causes for stragglers: PyTorch memory fragmentation and false kernel dependencies. While both issues had been reported before as potential performance problems, they were not known to lead to stragglers prior to our work.

6 Validation of Simulation Fidelity

Our analysis uses a simulator to estimate a job's execution time in the absence of stragglers, and also after fixing some stragglers. We validate the simulator's accuracy in two ways: 1) by comparing a job's average step time in the simulated original timeline to its average step time in the actual timeline; 2) by artificially injecting stragglers and comparing simulated and actual slowdowns.

Simulation inaccuracy and its sources. As we described in §3, not all types of operations are recorded in the trace: specifically, operations run on the CPU, including data loading, are omitted from the trace. In most cases, this limitation does not pose a problem because these CPU operations are overlapped with GPU ones, allowing their latency to be hidden. However, the overlap is not always perfect and delays the launch of the next operation. This *launch delay* is not simulated, constituting the main source of discrepancy in the simulation. To measure the discrepancy, we compute the average step run-time $\tau = T/n$ of the simulated original timeline (§3.3) for a job with n steps and compare it with the actual step run-time τ_{act}. In our experiments, the simulation discrepancy has a median of 1.3% and 90-percentile of 5.5%.

After examining the traces with large simulation discrepancies, we found three main causes: (a) As data is stored on a remote storage cluster using a separate, slower network, data loading is susceptible to network slowdown and time-out, leading to long launch delays for the first forward-compute operations of some training steps; (b) Long context jobs can have long delays before the launch of the first forward-compute operations of each step because, as part of batch preparation, samples are padded to the maximum sequence length, which is time-consuming; and (c) Early deployments of the planned GC optimization, where the GC was run every few training steps before gradient synchronization, slowed down the launch of those operations. We have resolved the latter two problems, but all three affect some of the traces that we captured. To ensure analysis fidelity, we dropped any traces where the simulation discrepancy is larger than 5%.

Validating the Accuracy of Slowdown Estimation. We validate the accuracy of slowdown estimation by running a job whose DP, PP, and TP degrees were 4, and CP degree was

1. We artificially slow down the first rank, *i.e.,* the worker with global-rank 0, by running a background process that periodically performs multiple matrix multiplications (MatMuls) of size 10K × 10K. By varying the time interval between MatMul launches, we create three levels of slowdown and measure the resulting slowdown. We also used the simulator to compute a simulated slowdown. Our measured slowdowns were 1.16, 1.40, and 2.03, respectively, which are close to the estimated slowdowns 1.21, 1.42, and 1.98.

7 Limitation

Next, we discuss the limitations of our analysis. Our limitations are due to the trace contents captured by NDTimeline, and we plan to address these limitations in future work.

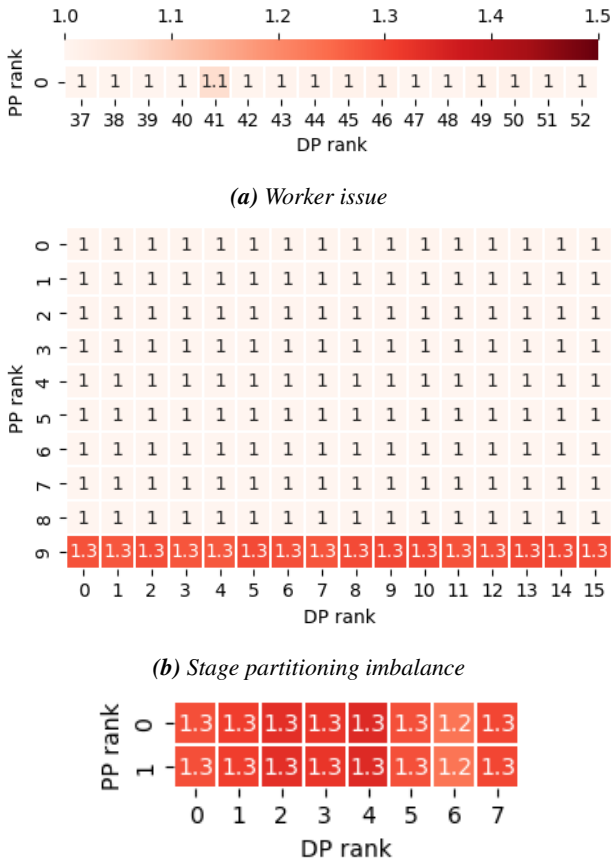
Limitations due to data captured by NDTimeline. NDTimeline performs coarse-grained profiling and records time taken by a microbatch's forward and backward computation. It is challenging to use this information to analyze stragglers that occur within a TP or CP group, because TP and CP groups synchronize, and stragglers at the TP or CP granularity show up as slow microbatches in our traces. As a result, if the straggler slows down all microbatches, our method of estimating idealized microbatch durations will not allow us to compute the ideal straggler-free execution time, and thus we cannot analyze the effect of such stragglers.

A second limitation is due to a bug in NDTimeline which affects some of our traces. This bug caused NDTimeline to not record some operations, resulting in simulating the launch of some forward- and backward-compute operations earlier incorrectly. The bug was spotted after we had begun collecting traces, and we post-processed traces affected by this bug to fix this problem.

Job covered by our analysis. To ensure analysis fidelity, we had to discard a portion of the traces we collected from our cluster. Consequently, our analysis does not cover all the LLM training jobs run within the cluster, and may underestimate straggler prevalence and severity. The reasons for discarding are mostly due to problems with the trace and we outline them below.

First, we eliminated any jobs that repeatedly fail. Our cluster uses a system that automatically resubmits jobs that fail. However, software bugs result in cases where a single job might fail dozens of times or more, and we eliminate these jobs to avoid introducing bias into our analysis. In particular, we discard any job that is restarted more than 15 times, which results in us discarding 13.9% of the recorded jobs or 7.3% of the recorded GPU hours.

Second, we discarded jobs for which we could not successfully run the what-if analysis. This results in us discarding 50.0% of the remaining jobs, which amounts to 34.1% of the remaining GPU hours. Most of these are due to issues with the trace: 28% of the traces were discarded because we could not parse the job's command line to determine the degree of parallelism; 28% were discarded because the job had too few steps to allow analysis;[7] and 25% were due to corrupt traces.



(a) Worker issue

(b) Stage partitioning imbalance

(c) Sequence length imbalance

Figure 14: *Heatmap patterns with different root causes.*

Finally, as noted earlier (§3.3), we discard jobs with large simulation discrepancy (> 5%), which amounts to 11.2% and 7.7% of the remaining jobs and GPU hours, respectively. This results in total coverage of 38.2% jobs and 56.4% GPU hours.

8 Online Straggler Detection and Diagnostics with What-if Analysis

To make it easier for users to benefit from the analysis we have discussed so far, we built an online monitoring service called *SMon* that runs automatically after each NDTimeline profiling session (which records dozens of training steps). SMon estimates slowdown, per-step slowdown, and worker slowdown, and presents the results on a webpage. Similar to Pingmesh [18], we present worker slowdowns using a heatmap, where each cell represents a worker with its x- and y-coordinate as its DP and PP rank, respectively, and the color depth represents the worker's slowdown. Our use of such a heatmap serves two purposes: first, it makes finding straggling workers easier, and second, the pattern of slowdowns often

[7] We filter out steps that are doing batch size warmup, and in some cases this results in jobs with insufficient steps.

helps pinpoint the initial root cause for the slowdown, making it easier to address the problem. For instance, jobs with worker issues, stage partitioning imbalance and sequence length variance have distinct patterns as shown in Figure 14. SMon also presents a per-step heatmap using per-step duration instead of average in Equation (4) when computing worker slowdowns, to reflect only the straggling within each step.

We have configured SMon to alert our on-call team whenever important jobs experience significant slowdowns. When alerted, the on-call team checks the worker heatmap to identify a suspected root cause by seeing if it matches one of the known patterns. We then use the per-step slowdown and the per-step heatmap to locate the problematic step and ranks to further drill down and understand the problem.

Within a month of its deployment, SMon has helped us identify and address several stragglers: it allowed us to locate faulty machines in three cases where machine problems were responsible for stragglers, it helped identify a case where sequence length variance was leading to large job slowdown, and a separate case where imbalanced compute partitioning across pipeline stages was to blame.

9 Related Work

Straggler in big data. The straggler problem has been extensively studied in big data frameworks such as MapReduce [3, 10, 16, 31], predating the era of deep learning. Mantri [3] provided an empirical characterization of stragglers in large-scale MapReduce jobs, where they quantify the straggler prevalence and attribute the root causes to load-imbalance, cross-rack traffic, and bad machines. They propose a system that actively restarts or duplicates outlier tasks while optimizing task placement and scheduling to mitigate delays. Ousterhout et al [31] conducted a more in-depth analysis of the performance bottleneck of Spark using a what-if simulation similar to our method. They attribute stragglers to diverse causes, including Java GC and file system delays.

However, the simulation approaches and findings of [3, 31] are not applicable to LLM training. First, much of the complexity for simulating MapReduce jobs is due to dynamic task scheduling, *e.g.,* a reduce task is scheduled to a machine after its inputs become available at runtime. In contrast, LLM training uses simple static scheduling, *i.e.,* tasks are placed at the beginning of the job and never changed. Second, the complexity of simulating LLM training jobs lies in handling the complex dependency structure stemming from mixed parallelism. In particular, LLM jobs have to account for pipeline parallelism, tensor parallelism, and data parallelism, each of which occurs at different granularities and can involve only a subset of ranks. By contrast, dependencies of MapReduce jobs are much simpler, *i.e.,* all map tasks communicate with all reduce tasks. These differences limit the direct applicability of their approaches and conclusions to LLM training.

Straggler in deep learning training. Straggler problems have also been studied in the context of data-parallelism in the deep learning era. Many of these prior proposals have focused on developing straggler mitigation strategies: Ramamoorthy et al [33] suggest using redundant coded computations to mitigate stragglers; Downpour SGD [9] and Project Adam [7] use asynchronous SGD and stale gradient updates to mitigate stragglers; Hop [27], Tensorflow [1] and Chen et al [6] use backup workers to resolve stragglers; and DropCompute [15] suggests dropping updates from stragglers. By contrast, our work focuses on characterizing the effect of stragglers and their root cause.

Straggler in LLM training. Recent studies have begun to explore stragglers in LLM training. Malleus [25] mitigates the impact of stragglers using dynamic parallelism adjustments based on real-time device performance. MegaScale [22], Llama3 [11] and Imbue's report [38] discuss infrastructure challenges for LLM training and briefly mention how they debug stragglers, but do not provide in-depth analyses. Unlike these works, we focus on a detailed and comprehensive analysis of the straggler problem itself.

FALCON [41] presents a detailed characterization of stragglers and introduces several mitigation strategies. Our work differs in several ways. First, we analyze jobs on a dedicated cluster while FALCON's analysis is for a shared cluster. The difference in the settings lead to different results: unlike FALCON, we did not see stragglers due to resource contention. Second, FALCON's analysis of large jobs (512 to 1024 GPUs) is only limited to 27 job traces, while our analysis has a significantly larger scale: among the 3079 traces we analyzed, 562 jobs use $\geq$ 512 GPUs. With only a few dozen traces, FALCON relies on manual analysis to study stragglers and determine their root cause. By contrast, we use a semi-automated approach combining simulation-based what-if analysis and manual verification of hypothesized root causes. Furthermore, FALCON's analysis overlooks stragglers that affect most steps rather than only a small fraction of steps. As we show in §4.2, the former is more common than the latter in our traces.

10 Conclusion

We present an in-depth study of the straggler problem in LLM training on job traces collected from our cluster. We use what-if analysis as our core approach, which allows us to estimate slowdown caused by stragglers as well as attribute it to different workers or operation types. These measurements help us study the overall impact of stragglers, characterize straggler symptoms from various aspects, and preliminarily diagnose the root cause. Through extensive case studies, we find that stage partitioning imbalance, sequence length imbalance, and Python's automatic GC are the main root causes; machine issues are rarely observed to cause stragglers, but they tend to cause severe slowdown when happening.

We have implemented parts of the analysis pipeline in SMon, a monitoring tool that is deployed in the training cluster at ByteDance. SMon allows us to more accurately detect

stragglers in running jobs and improves the diagnostics efficiency.

Acknowledgement

We thank OSDI reviewers and our shepherd for their insightful feedback. We also thank Lingfan Yu, Zhihao Zhang, Tao Wang and Lei Zhang for insightful discussions. Haibin Lin and Jinyang Li are the corresponding authors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, 2013.

[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in {Map-Reduce} clusters using mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[4] Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, Jie Tang, Yuxiao Dong, and Juanzi Li. Longalign: A recipe for long context alignment of large language models. *arXiv preprint arXiv:2401.18058*, 2024.

[5] Anupam Bhatnagar, Brian Coutinho, Xizhou Feng, Yifan Liu, Sung-Han Lin, Louis Feng, Michael Acar, and Yuzhen Huang. Holistic trace analysis. https://hta.readthedocs.io/en/latest/index.html, 2023.

[6] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[7] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pages 571–582, 2014.

[8] Charlie Curtsinger and Emery D Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[12] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[13] NVIDIA Forum. How many streams? maximum number of streams, 2022.

[14] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 178–194, 2024.

[15] Niv Giladi, Shahar Gottlieb, Moran Shkolnik, Asaf Karnieli, Ron Banner, Elad Hoffer, Kfir Yehuda Levy, and Daniel Soudry. Dropcompute: simple and more robust distributed synchronous training via compute variance reduction. *arXiv preprint arXiv:2306.10598*, 2023.

[16] Sukhpal Singh Gill, Xue Ouyang, and Peter Garraghan. Tails in the cloud: a survey and taxonomy of straggler management within large-scale cloud data centres. *The Journal of Supercomputing*, 76:10050–10089, 2020.

[17] Google. Perfetto. "https://perfetto.dev/".

[18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline

parallelism. *Advances in neural information processing systems*, 32, 2019.

[20] ByteDance Inc. Ndtimeline — vescale. "https://github.com/volcengine/veScale/blob/main/vescale/ndtimeline/README.md".

[21] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. Dynapipe: Optimizing multi-task training through dynamic pipelines. *arXiv preprint arXiv:2311.10418*, 2023.

[22] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[23] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

[24] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[25] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. Malleus: Straggler-resilient hybrid parallel training of large-scale models via malleable data and model parallelization. *arXiv preprint arXiv:2410.13333*, 2024.

[26] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023.

[27] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. Hop: Heterogeneity-aware decentralized training. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 893–907, 2019.

[28] Megatron-lm. https://github.com/NVIDIA/Megatron-LM, 2020.

[29] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[30] Kelvin KW Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 595–610, 2023.

[31] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[32] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[33] Aditya Ramamoorthy, Anindya Bijoy Das, and Li Tang. Straggler-resistant distributed matrix computation via coding theory: Removing a bottleneck in large-scale data processing. *IEEE Signal Processing Magazine*, 37(3):136–145, 2020.

[34] Chaofan Tao, Qian Liu, Longxu Dou, Niklas Muennighoff, Zhongwei Wan, Ping Luo, Min Lin, and Ngai Wong. Scaling laws with vocabulary: Larger models deserve larger vocabularies. *arXiv preprint arXiv:2407.13623*, 2024.

[35] NVIDIA Team. Context paralellism. "https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html".

[36] Python Team. Garbage collector design. "https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md".

[37] PyTorch Team. Pytorch distributed key-value store – tcp store. "https://pytorch.org/docs/stable/distributed.html#torch.distributed.TCPStore".

[38] The Imbue Team. From bare metal to a 70b model: infrastructure set-up and scripts. "https://imbue.com/research/70b-infrastructure/", 2024.

[39] Tianwen Wei, Bo Zhu, Liang Zhao, Cheng Cheng, Biye Li, Weiwei Lü, Peng Cheng, Jianhao Zhang, Xiaoyu Zhang, Liang Zeng, et al. Skywork-moe: A deep dive into training techniques for mixture-of-experts language models. *arXiv preprint arXiv:2406.06563*, 2024.

[40] Wikipedia contributors. Multiway number partitioning — Wikipedia. "https://en.wikipedia.org/wiki/Multiway_number_partitioning".

[41] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. FALCON: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training. *arXiv preprint arXiv:2410.12588*, 2024.

[42] Man Tsung Yeung, Penghui Qi, Min Lin, and Xinyi Wan. Balancing pipeline parallelism with vocabulary parallelism. *arXiv preprint arXiv:2411.05288*, 2024.

[43] Zili Zhang, Yinmin Zhong, Ranchen Ming, Hanpeng Hu, Jianjian Sun, Zheng Ge, Yibo Zhu, and Xin Jin. DistTrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models, 2024.

[44] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.

A Artifact Appendix

Abstract

This artifact provides the core functionality of the simulator and the what-if analysis proposed in the paper, including:

- estimation of overall slowdown S (Eq. 1);
- estimation of slowdown S_t attributed to each operation type t (Eq. 2);
- estimation of slowdown S_w attributed to each worker w (Eq. 4);
- computation of characterization metrics M_W (Eq. 5) and M_S for individual worker issues and stage partitioning imbalance, respectively;
- heatmap visualization as in Figure 14; and
- timeline generation of the simulated ideal trace visualizable in Perfetto [17].

It also provides three sample traces to demonstrate the usage of the tool. More details can be found in "README.md".

Scope

The artifact allows readers to validate if the provided implementation of the simulator and the what-if analysis match the description in the paper and if the code is functional and sufficiently documented.

Contents

The artifact contains the source code of the simulator and a set of proposed what-if analysis, three sample traces and their expected analysis result. It also contains a README file with detailed documentation of the project and step-by-step instructions to run the analysis using the traces as examples.

Hosting

Our artifact is hosted on GitHub at `https://github.com/ByteDance-Seed/StragglerAnalysis` on the `artifact` branch. The commit version is `8efcd7f`.

Requirements

There is no special hardware or software requirement for this artifact, except a somewhat recent version of Python. The code has been tested with Python 3.11.

Fork in the Road: Reflections and Optimizations for Cold Start Latency in Production Serverless Systems

Xiaohu Chai[1,2], Tianyu Zhou[2], Keyang Hu[1], Jianfeng Tan[2], Tiwei Bie[2], Anqi Shen[2], Dawei Shen[2], Qi Xing[2], Shun Song[2], Tongkai Yang[2], Le Gao[2], Feng Yu[2], Zhengyu He[2], Dong Du[3], Yubin Xia[3], Kang Chen[1], and Yu Chen[4,1]

[1]Tsinghua University, [2]Ant Group, [3]Shanghai Jiao Tong University, [4]Quan Cheng Laboratory

Abstract

Serverless computing has seen widespread adoption in public cloud environments. However, it continues to suffer from *long cold start latency*, which remains a key performance bottleneck. We have conducted an in-depth investigation of existing cold start optimizations and evaluated their effectiveness in large-scale industrial deployments. Our study reveals several common limitations in prior research: (1) reliance on simplified assumptions that overlook the complexities of large-scale systems; (2) a narrow focus on optimizing isolated components of the cold start process, while ignoring end-to-end workflow interactions; and (3) insufficient attention to the challenges introduced by concurrent execution environments. As a result, despite incorporating prior techniques, cold start latency on the Ant Group serverless platform remains in the range of hundreds of milliseconds to several seconds.

This paper identifies three previously overlooked sources of latency: (1) *control path latency*, stemming from interactions within the serverless runtime; (2) *resource contention latency*, arising under high concurrency and sustained execution; and (3) *user code initialization latency*, which reflects the trade-off between resource efficiency and startup performance. To address these challenges, we propose a suite of novel techniques that overcome key limitations in existing approaches. These techniques are designed to be both adaptable to real-world workloads and scalable to large deployments. Our system, ***AFaaS*** (short for Ant FaaS), reduces cold start latency to the millisecond level. AFaaS has been deployed in production for over 18 months and has consistently demonstrated stable performance at scale.

1 Introduction

Serverless computing is a cloud paradigm that has significantly transformed how applications are developed and deployed. By abstracting away infrastructure management, it enables developers to focus solely on application logic, while the serverless platform automatically handles resource allocation, scaling, and cost optimization. This paradigm has been widely adopted by major cloud providers, including AWS Lambda [10], Azure Functions [47], Google Cloud Serverless [36], Alibaba Serverless Application Engine [7], and Huawei Cloud Functions [39].

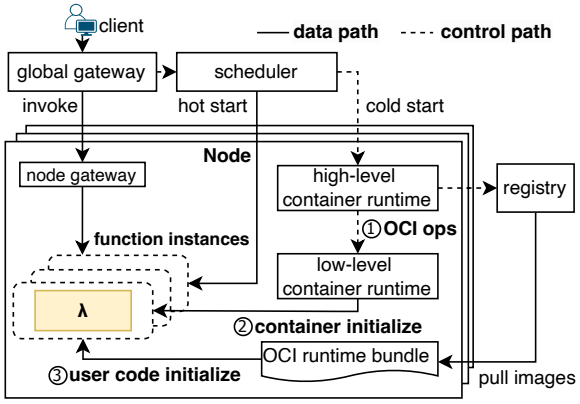


Figure 1: **Application start procedure in AFaaS.** This is simplified to show the key components of the OCI (Open Container Initiative) [41] specification.

However, the on-demand and pay-as-you-go nature of serverless computing introduces a persistent challenge: *cold start latency*. When a request arrives and no pre-warmed instance is available, the platform must initialize a new instance to run the target function—typically incurring a delay of over one second. Given that serverless functions often execute within just 50–100*ms*, such startup latency has become a well-known performance bottleneck in contemporary serverless platforms [6, 27, 28, 58, 96].

To address this challenge, researchers have proposed various optimizations. Checkpoint/restore techniques [12, 28, 89] and fork-based approaches [6, 27, 28, 68, 96] aim to bypass initialization by reusing state from pre-initialized templates. Caching-based methods [1, 2, 6, 11, 33] accelerate response time by reserving instances for frequently invoked ("hot") functions. Additionally, customized serverless runtimes [5, 58, 64] reduce cold start overhead by removing unnecessary features from the execution environments.

Figure 1 illustrates the current cold start procedure within

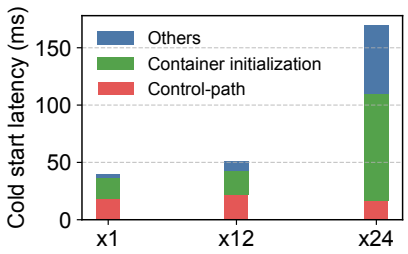


Figure 2: **Control-path costs.** Time spent on the control path when a function instance starts (excluding the initialization and execution of user code). $\times 1$, $\times 12$, and $\times 24$ represent different concurrency levels (*the number of functions that can be executed in parallel on a single serverless server*).

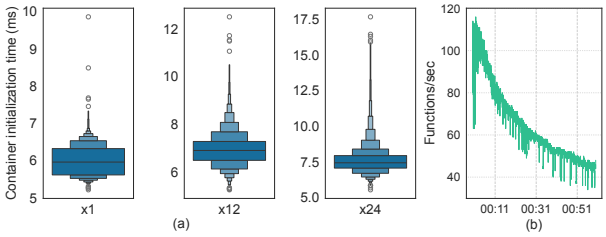


Figure 3: **Resource contention leads to unstable performance.** (a) Tail latency exhibits a clear positive correlation with concurrency levels, with ranges of $[5.21\text{ms}, 9.85\text{ms}]$ for $\times 1$, $[5.27\text{ms}, 12.49\text{ms}]$ for $\times 12$, and $[5.53\text{ms}, 17.67\text{ms}]$ for $\times 24$. (b) Under sustained high concurrency ($\times 24$ for 1 hour), throughput gradually decreased from an initial rate of 110 FPS (Functions Per Second) to 45 FPS.

a node, including three main phases: (1) *Control-path Interaction*: A high-level runtime (e.g., containerd) requests a low-level runtime (e.g., runc) to create a new instance, typically involving several binary calls and RPC operations based on the OCI (Open Container Initiative) [41] specification. (2) *Container Initialization*: The low-level runtime creates a function instance and completes its initialization — this has been the primary focus of previous research on cold starts in serverless computing. (3) *User Code Initialization*: After the function instance is ready, the instance must load and initialize the user code provided by the application developers.

Through real-world deployments, we identify three critical gaps in existing research on serverless cold starts. These factors significantly impact user-perceived latency but are widely overlooked. As a result, cold start latency on the Ant Group serverless platform remains in the range of hundreds of milliseconds to several seconds.

Gap-1: Control-path latency. The cold start involves more than just setting up the function instance itself; the control-path is equally critical. As shown in Figure 2, when the initialization of the function instance is optimized to a short duration, the latency on the control-path becomes prominent. Specifically, in our environment, the control-path latency takes 18*ms*–25*ms*, constituting approximately 30%–40% of the overall cold start period with Catalyzer [28]. The major source of these costs comes from the interaction between high-level and low-level runtimes, following the OCI specification [41]. For example, the shim-based invocation and binary call conventions (§2.2) incur overhead due to excessive interaction between multi-layer components.

Gap-2: Resource contention latency. Within the complex circumstances of the production environment, we witness performance instability. As illustrated in Figure 3, two scenarios are discerned: (1) Under high concurrency, the distribution of response time is significantly dispersed, presenting considerable tail latency. (2) During sustained execution, there is a marked decline in request throughput. Our profiling shows that the performance decline is primarily caused by resource contention within the host operating system during container startup. This issue has been largely overlooked in prior work and becomes especially critical under heavy workloads with intense contention.

Gap-3: User code initialization latency. Beyond initializing the container instance itself, the execution of user code introduces additional delays. For example, Just-In-Time (JIT) compilation or language runtime-specific code loading can significantly affect startup latency, especially when dealing with complex application frameworks or extensive dependencies (e.g., Spring [72], AngularJS [35]). User code initialization latency depends on user-specific behaviors. This makes it challenging for current optimization methods, such as provisioning [11] and SnapStart [12], to balance cost and efficiency.

Based on reflections from previous efforts and our experience in large-scale industrial environments, we develop a series of innovative techniques that overcome significant challenges to build a system, **AFaaS**, that adapts to real-world workloads and supports large-scale deployment.

First, we discover that the current cloud-native OCI [69] is not well-suited for serverless scenarios. Instead, we propose the Function Runtime Interface (FRI), which simplifies and optimizes the control path for function instances to reduce communication overhead between high-level and low-level components (§4.1).

Second, we identify bottlenecks caused by resource contention and conduct an in-depth analysis of the factors leading to performance instability. To address these issues, we propose strategies to mitigate contention. For inheritable and sharable resources, we integrate them into a ***seed*** (*sandbox template, a sandbox contains the initialized state to skip the initialization* [28]) to minimize redundant operations. For pre-allocatable resources, we use a pooling strategy to allocate them in advance, reducing concurrent conflicts. These strategies ensure consistent performance during concurrency and continuous execution (§4.2).

Finally, we extend the seed to include user code, preloading frequently invoked and time-consuming routines. To

optimize memory usage, seeds are managed using a hierarchical tree structure, where Copy-on-Write (CoW) is employed to efficiently create child nodes from the parent seed. This design reduces memory footprint while preserving fast startup performance (§4.3).

To our knowledge, AFaaS is the first commercial serverless system to achieve millisecond-level start latency. Our evaluation shows that AFaaS achieves a substantial reduction in end-to-end latency, ranging from 1.80× to 8.14× compared to the state-of-the-art Catalyzer in production scenarios, while maintaining a consistent start time between 5.45 ms and 9.41 ms. Under heavy workloads with high concurrency and sustained execution, AFaaS continues to deliver stable startup performance, with latencies between 6.97 ms and 14.55 ms, significantly outperforming Catalyzer's 38.39 ms to 74.05 ms. The system has been running reliably in production for over 18 months. All traces used in this paper are available at https://github.com/antgroup/AFaaS.

2 Characterizing Cold Start in Production Serverless Systems

2.1 Reflections for Cold Start in the Wild

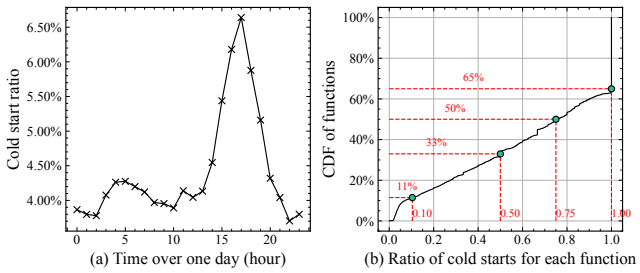


Figure 4: **Cold start pattern.** (a) The ratio of cold starts during a 24-hour period. (b) CDF of the cold start probability for each function.

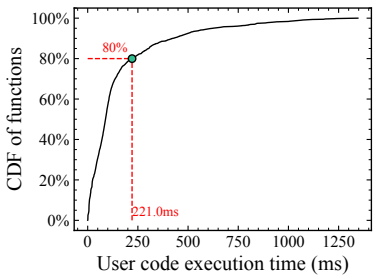


Figure 5: **User code execution time in Ant Group.**

Rapid Growth of FaaS in Ant Group. Currently, there are over 50,000 unique functions running on the public cloud in Ant Group, with approximately 100 million function calls per day. These functions are implemented in a variety of programming languages, including Node.js, PHP, Python, Java, and C++, and are executed on various runtime versions of each language. The scale is continuously growing.

Implication-1: Cold start is still significant as more than 50% of the functions exhibit a cold start probability exceeding 0.75. For security reasons, Ant Group deploys each function in a (micro) VM-based secure container [28,40]. Figure 4(a) illustrates the proportion of cold starts for a specific tenant over a given period in Ant Group. While over 95% of requests benefit from *hot start*, we observe that a significant portion of these *hot* requests are concentrated in a small number of frequently invoked functions (such disparities have also been observed in [44, 78]). As shown in Figure 4(b), more than 50% of the functions exhibit a cold start probability exceeding 0.75, while over 35% of the functions have a cold start probability of 1.

Implication-2: Caching-based hot start is insufficient to address the start latency issue (mainly) because of resource constraints. Like most cloud providers, to serve user requests more efficiently, Ant Group does not immediately destroy function instances after completing a request. Instead, these instances are retained for a period in the cache, enabling subsequent requests to be handled immediately if the function instances are still active. This process, known as a *hot start*, significantly enhances FaaS performance. AWS, Azure, and OpenWhisk keep containers alive for 10, 20, and 30 minutes, respectively, before terminating idle function instances [75,78]. However, caching infrequent functions would consume excessive memory. To achieve more flexible memory utilization, Ant Group limits hot instances to 1 minute, ensuring support for high-frequency functions. We still need to optimize the cold start performance, as over 60% of functions experience cold starts more frequently than hot starts.

Implication-3: Fork-based optimizations are essential but (still) insufficient to optimize end-to-end start latency. Figure 5 shows the user code execution time distribution in Ant Group— more than 80% of requests are processed within $221ms$, highlighting the importance of optimizing cold start latency. This observation differs from prior ones in [77]. To optimize cold start, we built secure container designs with the *sfork* (sandbox fork) mechanism of Catalyzer [28], which introduces optimizations to fork secure containers. This mechanism enables the creation of new function instances in $< 1ms$ under optimal conditions. Through long-term deployment and practical use of Catalyzer in production environments, several shortcomings have been identified. Firstly, Catalyzer primarily focuses on the fork operation of the sandbox process but overlooks key steps in the end-to-end container cold start sequence. These missing steps include the control path between the high-level runtime and the Catalyzer runtime, as well as the loading of user code after sandbox initialization. Secondly, in high-concurrency environments, resource contention leads to significant performance fluctuations during both the sandbox fork and initialization phases.

Revisiting End-to-End Cold Start Latency in Production.

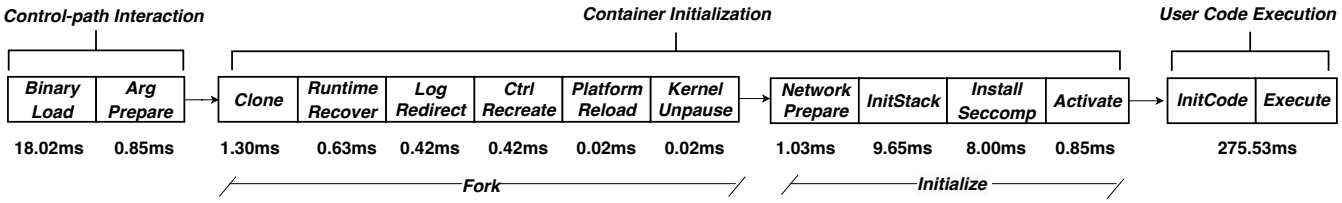


Figure 6: **End-to-end latency of cold start.** The numbers represent the latency of each phase of a Node.js-based function (*nunjucks*—rendering the template to HTML), detailed experimental setup can be found in §6. The entire end-to-end start-up process consists of three main stages: (1) ***Control-path Interaction.*** ① *BinaryLoad*: The runtime shim [22] loads the Catalyzer engine binary and starts the client process. ② *ArgPrepare*: The client process prepares fork-related parameters and invokes the seed sandbox. (2) ***Container Initialization.*** ① *Clone*: The seed sandbox invokes the *clone()* syscall to fork a new sandbox. ② *RuntimeRecover*: The forked sandbox initializes the Go runtime. ③ *LogRedirect*: The stdout and stderr of the sandbox are redirected. ④ *CtrlRecreate*: The sandbox recreates the control server. ⑤ *PlatformReload*: The sandbox transitions from VMX root mode to VMX non-root mode. ⑥ *KernelUnpause*: The guest OS is unpaused to resume execution. ⑦ *NetworkPrepare*: The client process prepares the network interface and creates a raw socket that binds to this network interface, then sends the raw socket to the sandbox. ⑧ *InitStack*: The guest OS initializes the network stack. ⑨ *InstallSeccomp*: The sandbox parses and installs a seccomp filter. ⑩ *Activate*: The sandbox is attached to the host cgroups and mounts the rootfs. (3) ***User Code Execution.*** ① *InitCode*: The sandbox loads the appropriate language runtime, compiles the user code, and also loads the function's dependency framework and libraries. ② *Execute*: The sandbox executes user code and provides a response.

End-to-end latency is a critical metric from the customer's perspective, yet it is often overlooked in existing works [61]. We define *end-to-end (E2E) latency* as the complete process encompassing the following steps: when a node receives a scaling request from the scheduler/auto-scaler, it initiates a cold start for a function instance, loads the corresponding code, retrieves the request from the node gateway, executes the function, and ultimately sends the response back to the end-user. Figure 6 illustrates the latency associated with each phase in a typical production function cold start using Catalyzer. The *end-to-end latency* consists of three main stages: *(1) The control-path interaction stage* involves command execution (e.g., sandbox fork) between high-level runtime (e.g., containerd) and low-level runtime (e.g., Catalyzer). *(2) The container initialization stage* sets up an isolated container from scratch. This stage involves two steps: first, cloning a new VM-process based on a template sandbox using the *sfork* mechanism [28], and second, installing container-related components such as namespace, virtual ethernet interface (veth), and seccomp rules for the VM-process. *(3) The user code execution stage* has undergone significant variation and typically includes initiating the language runtime (e.g., Python or PHP), initializing the code framework (e.g., Django [3] or Spring [72]), loading the function code and common libraries into memory for JIT compilation, and finally executing the user code.

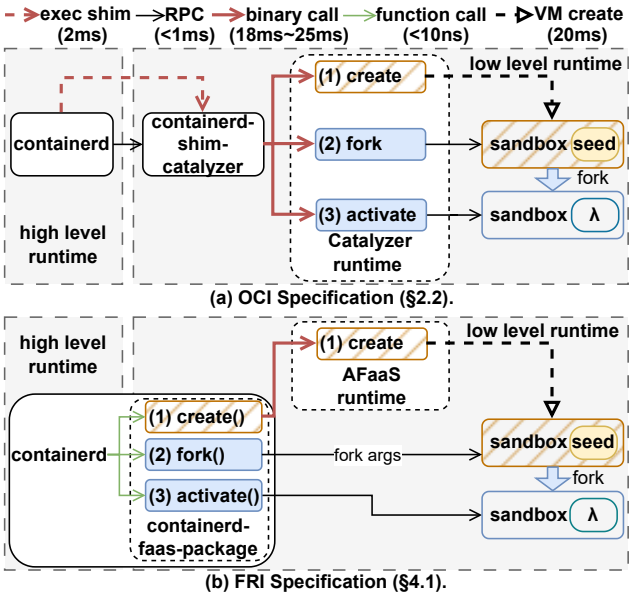


Figure 7: **Interactions between the high-level container runtime and the low-level container runtime.** Operations in the hatched boxes are only performed once during the creation of seeds.

2.2 Gaps from an Industry Perspective

Through real-world deployments, we identify three critical gaps overlooked by existing research on serverless cold starts that significantly impact user request latency.

Gap-1: Control-path latency. In containerized environments, interactions between high-level and low-level runtimes involve a long control path, introducing significant latency. When a creation request is received, the high-level runtime does not directly manage the instance. Instead, it spawns a runtime shim process to act as an adapter (containerd-shim-catalyzer), delegating the actual operations to this shim. Communication between the high-level runtime and the shim occurs via RPC requests. The shim uses the binary call convention [22], i.e., loading a runtime binary first (e.g., Catalyzer runtime) and then calling a function in the binary to manage the container. The loaded runtime binary, provided by the

lower-level runtime, is responsible for starting and stopping the container. This *shim+engine* architecture simplifies the integration of various runtimes, particularly those adhering to the OCI specification [30].

Though Catalyzer has optimized the creation of the sandbox for user code using the sandbox seed, avoiding creating from scratch, the extended invocation chain is still too long. As illustrated in Figure 7(a), the control path from the high-level runtime (containerd [23]) to the Catalyzer runtime primarily consists of three steps: (1) containerd communicates with the Catalyzer shim via RPCs. (2) Catalyzer shim loads and executes the Catalyzer runtime engine (one binary call for each serverless function invocation), and establishes communication with it via RPCs. (3) Catalyzer runtime engine creates the Catalyzer container instance, or triggers operations such as *fork()* and *activate()* through RPC calls. Though in a fork-based scenario, the seed containers are only prepared once during initialization, and subsequent operations are based on forking from the seed, ⓐ the costly binary loading process for *each serverless function invocation* still consumes 18.02*ms* (as shown in Figure 6). Also, ⓑ the use of the shim and Catalyzer runtime has extended the control path for high-level runtime (containerd) to create a sandbox for user code.

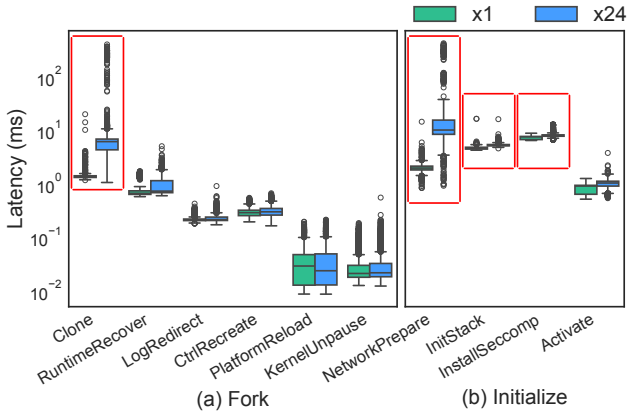


Figure 8: **Comparison of performance fluctuations caused by concurrent scenarios in key phases of container cold start.** The ones in the red boxes have longer delays and are the focus of our optimization.

Gap-2: Resource contention latency. Figure 3(a) shows significant latency fluctuations during each stage of container cold start under high concurrency. We perform a breakdown analysis of the latency fluctuations in different phases of container cold start at various concurrency levels, as shown in Figure 8. Compared to serial execution ($\times$1), cold start latency and fluctuations increase substantially during concurrent execution ($\times$24). We identify four key latency-intensive phases:

First, the latency of ***clone()*** operation during the sandbox fork stage degrades significantly under high-concurrency, increasing from 1.45*ms* to nearly 418*ms* in the worst case. The primary overhead comes from the *clone()* system call which requests new network and IPC namespaces. This process results in contention for host OS kernel locks. Through further profiling, we also discover that the performance degradation shown in Figure 3(b) is also related to the creation of IPC namespaces. IPC namespace creation is tied to POSIX message queues, as each IPC namespace requires a separate mount point for message queues. These mount points depend on VFS operations, including the lookup or allocation of superblocks. Although the Linux kernel employs a superblock caching mechanism, the cache hit rate is relatively low because each IPC namespace usually requires a unique superblock. Under high concurrency scenarios, contention over global locks (such as *sb_lock*) and mount locks (like *mount_lock*) increases synchronization overhead, further exacerbating performance degradation.

Second, the **network preparation** phase exhibits substantial latency degradation (from 2*ms* to 21.14*ms*). During this phase, Catalyzer engine parses the *config.json* file provided by containerd, switches into the specified network namespace via *setns()*, profiles the network interface's IP address, route, and device index, and creates a raw socket binding to this network interface. After setting up the raw socket, Catalyzer engine exits the network namespace using *setns()*. These operations involve intensive interactions with Linux kernel resources and impose a high CPU load, leading to contention and performance fluctuations in concurrent scenarios.

Third, in the **network stack initialization** phase, the sandbox instance uses the raw socket *fd* passed from the Catalyzer engine through RPC to initialize the container's network stack, which involves creating protocol stack ring buffers and initializing complex data structures. This process incurs substantial memory access and frequent VM exits to the host OS, leading to the latency fluctuations from 4.82*ms* to 6.94*ms*.

Last, **the seccomp installation** phase experiences several-fold latency spikes under high concurrency. Seccomp filters (seccomp-bpf) are used to restrict the system call privileges of the sandbox process to enhance container platform security. This phase involves parsing seccomp configurations, compiling seccomp rules, and installing the compiled seccomp rules into the kernel, leading to contention for host kernel locks and resulting in the latency fluctuations from 7.58*ms* to 13.01*ms*.

Gap-3: User code initialization latency. User code initialization involves loading the language runtime (e.g., Node.js, Python, Java), compiling the code, and loading dependent libraries. These phases are time-consuming, often exceeding the function's execution time and significantly impacting cold start latency. For example, in a Node.js-based function, 275.53*ms* of the user code execution phase includes 238.73*ms* spent on loading and initializing dependencies.

2.3 Cold Start Optimizations

By revisiting prior cold-start optimization approaches in the context of the industry challenges discussed in §2.2, we derive key insights that guide the design of AFaaS, as summarized in

Table 1: **Review of prior work and our design in AFaaS.**

Code Path	Prior Work	Design in AFaaS
Control Path	*Liu et al.* [61] surveys orchestration bottlenecks. *Dirigent* [26] proposes inter-node orchestration strategies. **Analysis:** Both overlook intra-node orchestration.	**FRI**: An efficient intra-node control path that replaces OCI (§4.1).
Clone	*Catalyzer* [28], *Molecule* [27], and *MITOSIS* [96] leverage fork/clone mechanisms to accelerate container startup. **Analysis:** Performance degradation arises from host resource contention, such as network and IPC namespaces, under high concurrency or sustained execution.	An enhanced cloning mechanism with pooling and sharing enables faster cold starts under high concurrency and sustained execution while preserving security (§4.2).
Network	*PCPM* [65] uses pause containers to retain and reuse the network stack, including IP configuration, virtual Ethernet (veth), and namespaces. **Analysis:** Sharing veth is an effective practice but with limitations: (1) it only supports traditional containers, as supporting secure containers incurs high memory overhead, and (2) it introduces security risks by reusing the TCP/IP stack from previous containers.	Lightweight veth sharing is achieved through pre-created, independent veth pairs; each fork initializes a clean TCP/IP stack to eliminate security risks (§4.2.1).
Cgroup	*RunD* [58] and *TrENV* [38] reuse cgroups with reconfiguration. **Analysis:** Cgroup reuse is an effective practice and is adopted in AFaaS.	Utilize a cgroup pool to assign and recycle a cgroup for each instance (§4.2.1).
Seccomp	*Catalyzer* [28] and *Firecracker* [5] install a strict seccomp filter to enhance security. **Analysis:** Parsing seccomp configurations, compiling seccomp rules, and installing the compiled rules each time can lead to a slowdown under high concurrency.	Seccomp rules are pre-compiled during seed preparation (§4.2.2).
User Code Initialization	**(1) *fork*:** *SAND* [6] and *Catalyzer* [28] share initialized code (e.g., language runtimes and loaded libraries) via *fork*. **Analysis:** *SAND* does not support multi-process runtimes (e.g., Java, Node.js). Catalyzer's seed can include user code. However, supporting user code fork in Catalyzer is inefficient since each seed consumes a large amount of memory. **(2) *C/R* (Checkpoint/Restore):** *AWS SnapStart* [12], SEUSS [19], and *Sabre* [56] leverage *C/R* techniques to reduce cold start latency. **Analysis:** *SnapStart* achieves only sub-second performance [12]. Unikernels [19] and hardware acceleration [56] face significant hurdles for adoption in production. **(3) Caching:** *RainbowCake* [99] adopts layer-wise pre-warming and keep-alive strategy. **Analysis:** A practical approach, but unfriendly to infrequent functions due to user template downgrades, leaving the cold start issue unresolved. Reusing the same language container also poses data leakage risks. **(4)** *TrENV* [38] and *MITOSIS* [96] share initialized code across nodes via RDMA/CXL. **Analysis:** A reasonable approach, but it requires specialized hardware that is not available in AFaaS. Deploying such hardware would also increase deployment costs.	Tree-structured seeds reduce overall resource usage, while multi-level forking offers best-effort performance with acceptable latency (§4.3). All instances are cloned from a clean-state seed and destroyed after execution for security enhancement (§5).

Table 1. While AFaaS incorporates several proven techniques, such as cgroup reuse [38, 58], our analysis also reveals three fundamental limitations in existing solutions.

Limitation-1: Insufficient Full-Stack Optimization for End-to-End Latency. While prior work focuses on optimizing specific cold-start stages (e.g., container initialization), it often overlooks systemic bottlenecks like control path overhead. Although inter-node orchestration issues have been explored [26, 61], intra-node control path inefficiencies remain largely unaddressed.

Limitation-2: Inconsistent Performance under Stress. Real-world deployments require strict adherence to SLOs [62]. Although recent efforts (e.g., recycled network stacks [65]) partially mitigate kernel-induced degradation, they still overlook critical bottlenecks, such as resource contention during fork/clone [27, 28, 96] and the latency of seccomp rule compilation [5, 28] under high load, leading to unstable tail latency.

Limitation-3: Intensified Complexity in Code Reuse. To achieve low-latency starts, FaaS platforms often cache specific function states for reuse (e.g., checkpoint/restore or fork) [11–13, 28, 99]. However, supporting the caching of user-level code presents challenges, including high memory requirements [11, 28], the need for specialized hardware [38, 56, 96], and vulnerable deployment strategies [99].

3 AFaaS Overview

System Architecture. Figure 9 shows the AFaaS architecture. First, there is a *node gateway* to receive requests from global control plane components (e.g., schedulers), forward the requests to the node's components like high-level runtime or function instances, and respond to external requests. Second, similar to other serverless platforms, the AFaaS platform deploys two levels of runtimes: a *high-level runtime* (containerd), and a *low-level runtime* (AFaaS). The high-level runtime will directly handle requests from the node gateway, while the low-level runtime focuses on managing secure containers. We implement our secure containers based on Catalyzer [28]. Notably, to resolve Gap-1, the high-level runtime and low-level runtime communicate efficiently through a new interface called FRI (Function Runtime Interface). Moreover, AFaaS employs strategies of *resource pooling and sharing*.

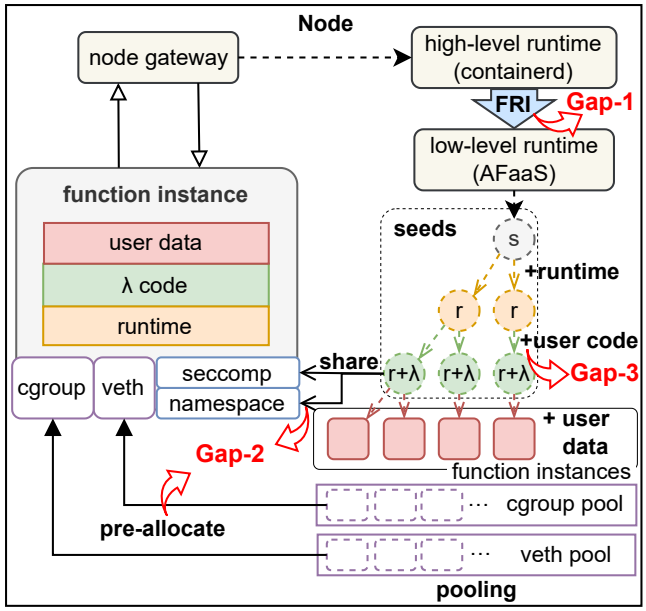


Figure 9: **AFaaS Design.**

Resources related to container allocation are managed to mitigate resource contention latency (Gap-2). Lastly, each node will maintain a tree-structured *seeds* (the template instance of functions) with different levels to resolve Gap-3, balancing performance and memory overhead. Next, we present three key insights that address the gaps identified earlier.

Insight-1: Many of the cross-runtime interactions are shim-calls that can be safely optimized. For Gap 1, we analyze the interaction requirements between high-level runtime and low-level runtime and propose FRI, a general bespoke control interface for serverless functions to shorten the control path (§4.1). A key observation is that cross-runtime interactions can be classified into two types: *shim-calls* (forwarding requests to another component) and *service-calls* (handling requests). For example, in the case of the *fork()* interface, the low-level runtime simply forwards the invocations from high-level runtime to the sandbox but does not perform any specific tasks, which is one of the shim-calls [22]. In contrast, the low-level runtime will perform complex work for *create()* to create a sandbox, which is classified as a service-call in our system. Based on this observation, we choose to replace the shim-calls, which are implemented by binary calls (18.02ms) now, with function calls by adopting a plugin-based method. The control path latency can be reduced accordingly.

Insight-2: Contended resources are more suitable to be reused or shared, rather than being created from scratch. For Gap 2, we observe that contended resources are usually low-level system resources; although their creation is not easily contention-free, they can be effectively reused and shared among instances. We adopt a resource pooling strategy for resources like cgroups and veth pairs to minimize allocation latency and mitigate resource contention. For resources that can be prepared in advance, such as seccomp filters and namespaces, we integrate them into the seed. When forking a new instance, these states can be inherited and reused directly [28, 96].

Insight-3: Fork can achieve both latency-efficient and resource-efficient with multi-level templates (or seeds). For Gap 3, we extend the seed to include user code, i.e., incorporating frequently used and time-intensive functions as templates. To manage the resource overhead of numerous seeds, we implement a hierarchical tree structure, using CoW to efficiently create child seeds from parent seeds and reduce memory usage (§4.3).

4 Detailed Design

In the following sections, we present detailed optimizations addressing each identified gap.

4.1 Shortening Control Path (Gap-1)

Current cloud-native standards, such as OCI [30], are tailored for long-running workloads. The numerous modules and lengthy control path defined by this standard make it ill-suited for serverless scenarios with high-concurrency startups and short container lifecycles. Catalyzer [28] enhances container startup by leveraging fork-based technology in the runtime engine. However, it still introduces a delay of 18*ms*–25*ms* due to executing the Catalyzer runtime engine binary, which sends an RPC to trigger the fork operation in the sandbox seed. This delay diminishes the impact of Catalyzer's sub-millisecond startup speed.

Function Runtime Interface (FRI). AFaaS introduces a lightweight serverless interface, *FRI*, to reduce the control-path latency between the high-level and low-level runtimes. The key innovation of FRI is the use of a *plugin-based interaction* model, replacing binary calls with direct function calls for frequently invoked interfaces in the serverless platform. Specifically, the low-level runtime implements the FRI interface, and the integrated plugin inside the high-level runtime interacts with it via RPC calls. As shown in Figure 7(b), we implement a *containerd-faas-package* plugin for containerd, which includes three main methods: *create()*, *fork()*, and *activate()*. The *create()* method instantiates a sandbox by loading the low-level runtime binary. Only the root seed (*level-0 seed*, §4.3) requires creation via *create()*, and this step is performed only once. After the root seed is created, the AFaaS runtime is ***never*** needed again, and the seed hierarchy is built accordingly. Subsequently, *fork()* and *activate()* are used to fork and activate new sandbox instances to run user code, which do not need to load the runtime binary as AFaaS will invoke these interfaces directly from the high-level runtime via RPCs. Thus, control latency is reduced due to ⓐ the avoidance of loading the AFaaS runtime binary each time a new serverless function is invoked, and ⓑ the shortened path where the high-level runtime directly invokes a corresponding seed to create a new

sandbox for user code. Furthermore, this design enables resource pooling for veth (§4.3), as AFaaS can reuse network resources by sharing the network namespace (netns) and reclaiming the network interface after the sandbox exits. The high-level runtime retains the netns and interface information and passes them via fork arguments.

Decision: *FRI demonstrates that strategic interface specialization for FaaS — sacrificing little modularity versus OCI — unlocks significant efficiency gains in cold starts, a transformative trade-off for serverless systems.*

4.2 Resource Pooling and Sharing (Gap-2)

To reduce resource contention latency, we design two *fork-friendly optimizations* without breaking the isolation guarantee of secure container (security analysis provided in §7): (1) *Resource Pooling*: Resources that can be pre-allocated are allocated or recycled in advance and stored in pools, reducing concurrent conflicts. (2) *Resource Sharing*: Resources that can be inherited and shared from the seed are integrated into the seeds, avoiding unnecessary reallocations.

4.2.1 Resource Pooling

Virtual Ethernet. AFaaS assigns a network device to each container for isolation. To minimize overhead, we pre-allocate a sufficient number of network devices, which are dynamically assigned during container creation and recycled once the container exits. This approach eliminates the significant overhead of network device creation.

Cgroup. Cgroups are used to limit container resource usage. In traditional cloud-native environments, new cgroups are created for each container and removed after the container is destroyed. This management mechanism can be time-consuming under high concurrency. AFaaS introduces a cgroup pool that recycles cgroups when the sandbox instance is deleted. When creating a new instance, an available cgroup will be selected from the pool and assigned to the instance.

4.2.2 Resource Sharing

Namespaces. Linux namespaces isolate containers on the host OS. In Catalyzer, each fork creates and configures new network and IPC namespaces, causing significant resource contention under high concurrency and sustained execution. AFaaS allows seeds and function instances to share the same network and IPC namespaces. Cross-VM instance sharing is secure, as user code runs within the guest OS of secure containers, ensuring isolation and preventing interference.

Seccomp. Seccomp is a Linux kernel feature that restricts the system calls a process can make. Secure containers enforce strict seccomp rules to prevent attacks, but installing seccomp involves parsing configurations, compiling rules, and installing them into the kernel, leading to contention and performance fluctuations. In practice, seccomp rules are generally fixed, as seen in gVisor [40] and Firecracker [5], which use predefined rules. In AFaaS, seccomp configuration parsing and compilation are completed during the seed preparation phase, so that when a new instance is forked, only a simple installation step is required.

Network Stack. The network protocol stack requires substantial memory for ring buffers and the initialization of complex data structures. Since each instance requires its own configuration, sharing the network stack is challenging. However, we observe that most operations are similar across instances. To avoid redundancy after forking, we divide the network module into shareable and non-shareable parts. The shareable components include key TCP/IP stack elements like clock, random seed, protocol handlers, TCP/UDP control blocks, and ARP tables. Non-shareable components, such as IP/MAC addresses and backend devices, must be assigned separately. Memory allocation and structure initialization are performed during the seed preparation phase. After forking, the raw socket is passed to the instance, which is then probed, bound to the veth interface, and the network stack is initialized.

Decision: *Contention bottlenecks are eliminated by decoupling key resources: pooling/sharing high-contention elements while retaining the sandbox fork's low-latency initialization, achieving faster cold starts in high-density deployments without sacrificing security.*

4.3 Seeding User Code (Gap-3)

To reduce application initialization overhead, Catalyzer uses language runtime templates with pre-initialized runtimes (e.g., Python, Node.js, JVM). As discussed in §2, initializing the runtime alone is insufficient for production workloads. User code initialization latency is influenced by factors such as dependent library loading, JIT compilation, and framework initialization (e.g., parsing large XML configuration files in Spring). In AFaaS, frequently invoked functions are optimized with dedicated seeds containing user code to reduce cold start latency. However, maintaining numerous container seeds can increase memory overhead. To balance performance and resource usage, AFaaS employs tree-structured seeds and multi-level fork mechanism.

Tree-structured Seeds. As illustrated in Figure 10, function seeds in AFaaS are organized hierarchically in a tree structure; the child seeds reuse the memory of their parents. During seed preparation, the high-level runtime performs a hierarchical search to select an appropriate seed for forking. The newly forked instance completes incremental tasks (e.g., language runtime initialization, dependency imports, JIT compilation) and then enters a paused state, becoming a new seed. Incremental updates are managed using a CoW mechanism.

Multi-Level Fork. When a new user request requires a func-

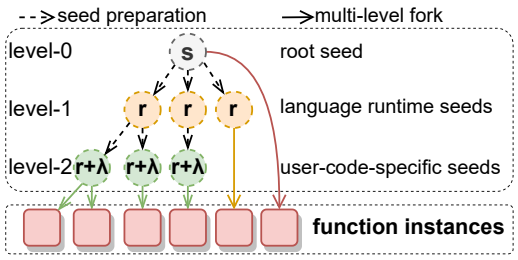


Figure 10: **Seed organization of AFaaS.** There are three types of seeds. The level-0 seed (root seed) contains data from the guest OS and is unique to each computing node. The level-1 seeds (language seeds) are forked from the level-0 seed and initialize the language runtime. The level-2 seeds (user-code-specific seeds) initialize the framework, import libraries, and compile user code.

tion instance, the high-level runtime searches the seed tree for a suitable seed. If a seed with the user code exists, it forks a container from that seed. Otherwise, it searches higher-level nodes to locate seeds with the language runtime, which typically fulfill the request.

Benefits. Our design offers two distinct advantages compared to previous approaches [11, 12] for caching user code: (1) *State Reusing*: The tree structure of seeds enables the sharing of physical memory pages across levels and nodes, leading to a reduction in global resource consumption; (2) *Best-effort fork*: In contrast to previous methods limited to forking from user-code seeds or starting from scratch, our multi-level fork approach based on the seed tree allows for best-effort provisioning. Even if a function lacks user-code-specific seeds, it can still be served with language runtime seeds or root seeds, ensuring acceptable performance.

Decision: *Tree-structured seeds with multi-level forking enable adaptive cold starts — initializing instances from the nearest viable ancestor to reduce resource overhead while maintaining low latency via granular state reuse.*

5 Implementation

We implement AFaaS based on Catalyzer [28] with the following enhancements for production.

Container Early Destroy. Container shutdown is also a critical concern, as the procedure can take over 10 seconds due to factors such as language runtime shutdowns (e.g., JVM), file system flushing, and kernel exit. AFaaS addresses this with container early destroy. Since serverless functions execute user code through a unified handler, the function is considered complete once the handler returns. At this point, AFaaS pauses the guest OS, disconnects TCP connections, and shuts down the container to reclaim resources. Due to the inherently ephemeral and stateless nature of functions in serverless computing, ensuring state correctness is relatively straightforward. For example, database access is typically performed via remote calls to centralized storage. By simply and properly terminating all TCP connections, the integrity of data consistency can be effectively safeguarded. Most production functions adhere to this pattern, enabling early destroy without affecting execution correctness.

EPT Prefill. EPT Prefill is used to reduce overhead from Extended Page Table (EPT) violations. The EPT translates Guest Physical Addresses (GPA) to Host Physical Addresses (HPA). When a seed forks a new instance, the instance transitions from root to non-root mode before executing user code. Although the seed and instance share physical memory pages, the EPT page table is uninitialized, leading to EPT violations when the guest OS accesses memory. Each violation triggers a VM-exit for page table reconstruction, causing significant overhead due to the short execution time of serverless functions. To address this, AFaaS employs EPT Prefill. During the fork, the seed copies its EPT page table to the new instance and marks the last level of the page table directory as read-only. This prevents EPT violations for memory reads.

6 Evaluation

Our evaluation aims to answer the following questions:

- *Q1:* How much improvement does AFaaS offer in reducing end-to-end latency and enhancing performance stability under heavy workloads? (§6.2)
- *Q2:* How does each technique in AFaaS optimize the end-to-end latency and performance stability? (§6.2 & §6.3)
- *Q3:* How does performance scale with varying levels of concurrency? (§6.4)
- *Q4:* How much memory can be saved by organizing FaaS instances in a tree structure? (§6.5)
- *Q5:* How is the performance of the production system after deploying AFaaS? (§6.6)

6.1 Methodology

Experimental Setup. We conduct all our experiments on an x86-64 machine with 24 cores Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz server and 512GB of RAM. The server runs Linux 5.10. The *wrk benchmarking tool* [91] is used to test end-to-end latency with varying concurrent requests. The evaluation takes the following systems as baselines.

1. ***CataOnly*** is the Catalyzer system augmented with our enhancements and optimizations described in §5. This configuration establishes the baseline serverless platform that exclusively leverages Catalyzer's capabilities while deliberately excluding other platform-level optimizations.
2. ***CataOPT1*** [1] represents an upgraded version of Catalyzer, to deal with resource contention latency (**CataOPT1 =**

[1] In practice, we implemented the optimization for Gap-2 first to reduce the container initialization latency, similar to that in related works.

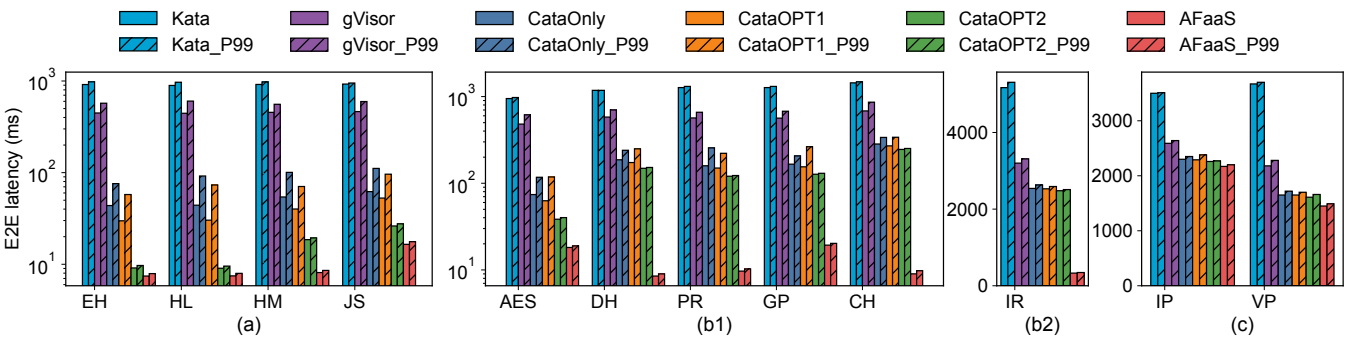


Figure 11: **End-to-end latency (sequential).** The hatched bar represents P99 latency, and the un-hatched bar indicates average latency. To enhance clarity, a logarithmic scale is employed for (a) and (b1), whereas a linear scale is utilized for (b2) and (c).

CataOnly + Gap2-OPT).

3. ***CataOPT2*** utilizes the FRI mechanism to optimize control path (**CataOPT2 = CataOPT1 + Gap1-OPT**).
4. ***AFaaS*** is our final production platform that further optimizes the user code initialization (**AFaaS = CataOPT2 + Gap3-OPT**).
5. ***Kata*** (v3.15.0) initializes by cloning from a pre-created VM (Kata-template [46]).
6. ***gVisor*** (release-20250414.0) starts from scratch [40].

We select Catalyzer as our baseline for its state-of-the-art sub-millisecond startup performance for secure containers. We also include gVisor and Kata as baselines because they are production-proven and open-source, and support the same hardware configurations.

Functions Evaluated. We select functions from representative serverless benchmarks, such as Function-Bench [49] and SeBS [25], which encompass a diverse range of scenarios. These include `EH` (Echo: response without any operation), `HL` (Hello: print "Hello, World"), `HM` (High Memory: malloc 1GB memory), `JS` (JSON Serde: deserialize and serialize JSON), `AES` (PyAES: encrypt messages), `DH` (Dynamic HTML: generate dynamic web pages), `PR` (PageRank: PageRank algorithm), `CH` (Chameleon: render HTML tables), `IP` (Image Processing: apply image processing algorithms to an image), `VP` (Video Processing: convert a video into grayscale), and `IR` (Image Recognition: image recognition using ResNet).

6.2 End-to-End Latency

End-to-End Latency. We first assess the average end-to-end execution time of typical functions in a sequential mode, as shown in Figure 11. Each function was tested for a duration of 1 minute. Based on the user code initialization and execution durations, we classify these typical functions into three distinct groups.

Figure 11(a) illustrates the functions with both short user code initialization and execution time. For these functions, the latency of control-path and container initialization accounts for a significant proportion. AFaaS achieves a speedup of 3.76×–6.68× for average latency and 6.31×–11.74× for P99 latency, compared with CataOnly. These improvements are largely attributed to the reduction in the control-path time facilitated by the FRI mechanism, coupled with optimizations during the container initialization phase. The differences between CataOPT2 and AFaaS are not significant.

Figure 11(b1/b2) illustrates functions that have longer initialization time for user code but with shorter execution time. This pattern is observed in functions that necessitate the one-time loading of large datasets or complex initialization operations. The performance enhancement of the AFaaS is remarkable. It pre-initializes user code in seeds, facilitating subsequent requests to execute promptly through forking. Compared to CataOnly, CataOPT2 reduces average latency by 1.02×–1.92× and P99 latency by 1.04×–2.92×. AFaaS reduces average latency by 4.09×–31.48× and P99 latency by 6.19×–34.51×.

Figure 11(c) illustrates functions with longer execution time. The performance optimizations of AFaaS in control-path, container initialization, and user code initialization are less noticeable compared to the function execution time. The performance improvements are about 1.05×–1.14× on average latency and 1.07×–1.15× on P99 latency.

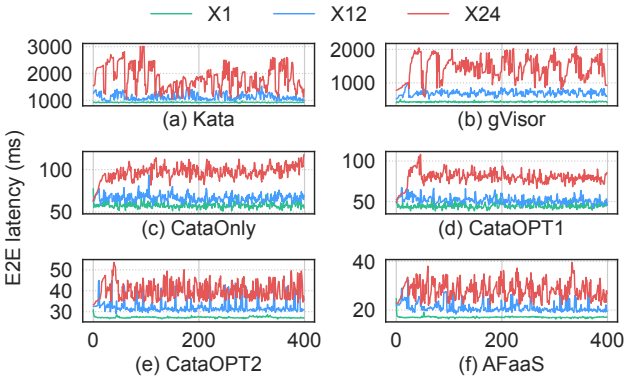


Figure 12: **E2E latency under different levels of concurrency.** 400 continuous samples for each system.

Concurrency and Sustained Traffic Performance. We evaluate the performance stability of AFaaS under different con-

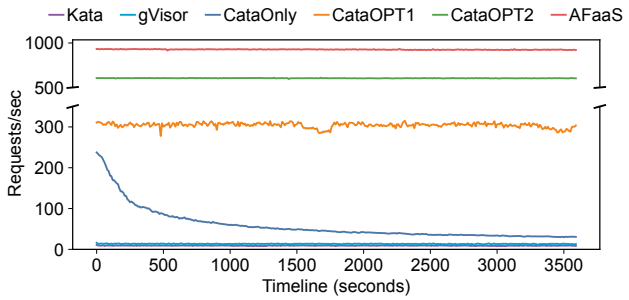


Figure 13: **Throughput variation over time at high sustained concurrency (×24).**

currency levels and sustained traffic using the `JS` function. Figure 12 contrasts the E2E latency and fluctuations under diverse levels of concurrency using `JS`. At concurrency levels ranging from 1 to 24, AFaaS demonstrates superior E2E latency and stability. The E2E latency of AFaaS falls within the range of [16.34 ms, 39.56 ms], compared to CataOnly's latency range of [51.32 ms, 117.92 ms]. Furthermore, the cold start time for AFaaS ranges from 6.97ms to 14.55ms, while for CataOnly, it spans from 38.39ms to 74.05ms. Kata and gVisor exhibit orders-of-magnitude worse E2E performance under high concurrency.

Figure 13 shows a notable decline in CataOnly's performance over time. The degradation is attributed to superblock cache misses and kernel lock contention (analyzed in §2.2), leading to a transition from the fast path to the slow path for newly created IPC namespaces, while other versions maintain relatively stable performance.

6.3 Optimization Breakdown

This section examines how CataOPT1 improves latency compared to CataOnly. The JS test function is used.

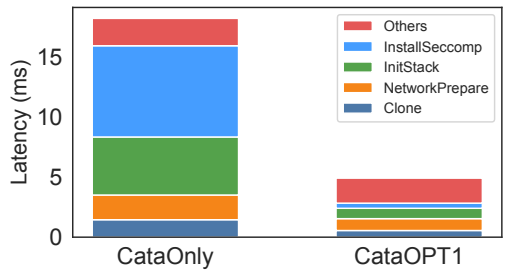


Figure 14: **Latency of CataOPT1 and CataOnly.** The "Others" category includes the remaining phases in the container initialization stage shown in Figure 6.

End-to-End Latency Optimizations. Figure 14 illustrates the optimization across various stages in sequential mode, highlighting key improvements from the *Clone*, *NetworkPrepare*, *InitStack*, and *InstallSeccomp* phases. In the *Clone* phase, Catalyzer invokes *clone()* with `CLONE_NEWNET` and `CLONE_NEWIPC`, placing the new instance into an independent network namespace (netns) and IPC namespace. CataOPT1 reduces namespace creation latency by sharing the namespace with the seed. During the *NetworkPrepare* phase, AFaaS minimizes the latency involved in parsing the specified netns path in *config.json* and executing *setns()*. In the *InitStack* phase, AFaaS decreases the initialization time of the network protocol stack. Finally, in the *InstallSeccomp* phase, AFaaS reduces the latency required to compile and load the seccomp rules.

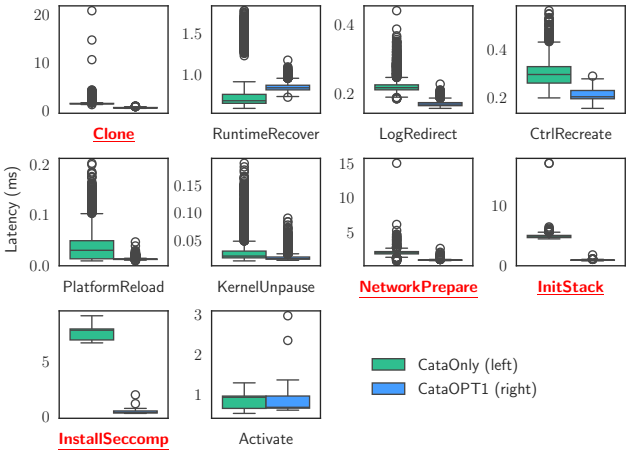


Figure 15: **Latency fluctuation of different phases during container initialization (10 phases in Figure 6).** Evaluated under sustained high concurrency (×24). The underlined phases are optimized.

Resource Contention Optimizations. We measure the fluctuation of various phases in the container initialization phase of CataOnly and CataOPT1 under high sustained concurrency (×24), as shown in Figure 15. Sharing the namespace reduces fluctuation during the *Clone* phase. The veth pool and FRI can decrease the fluctuation in parsing *config.json* during the *NetworkPrepare* phases. Pre-initialization of the protocol stack can mitigate the fluctuation in the *InitStack* phase, and the use of seccomp pre-compile technology can reduce the fluctuation associated with seccomp rule compiling and loading. After mitigating the main resource contentions, fluctuations in other phases are also reduced.

6.4 Scalability

We evaluate the throughput and end-to-end latency of different systems as the degree of concurrency gradually increased with the `JS` function. The results in Figure 16 show that the throughput of AFaaS scales significantly with increasing concurrency, demonstrating the best scalability. AFaaS also maintains the lowest end-to-end latency across all concurrency levels, with the least gradual increase. For comparison, CataOnly shows relatively high end-to-end latency and quickly hits a performance bottleneck, with a slight decline as the number of

concurrent requests increases. Both Kata and gVisor exhibit poor performance in terms of throughput and latency.

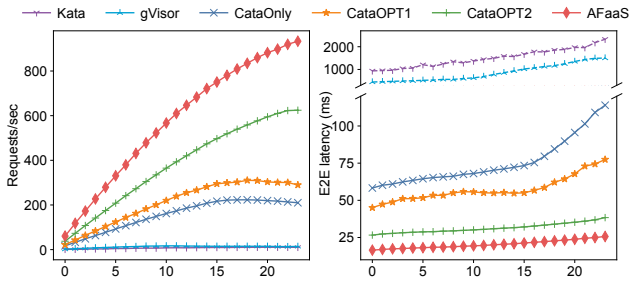


Figure 16: **Throughput (left) and E2E latency (right) with varying levels of concurrency.**

6.5 Memory Consumption of Multi-Level Fork

As shown in Figure 17(a), for seeds with the same user code, AFaaS reduces memory usage by 28.11%–84.91% compared to CataOnly. This is because the seeds share the *guest OS* and *language runtime*. For functions with long initialization time and a high execution volume, this approach is both cost-effective and efficient. When the user code size is large (e.g., `VP`, `IR` in Figure 17(b)), the shared components have a minimal impact, and the benefit is less noticeable.

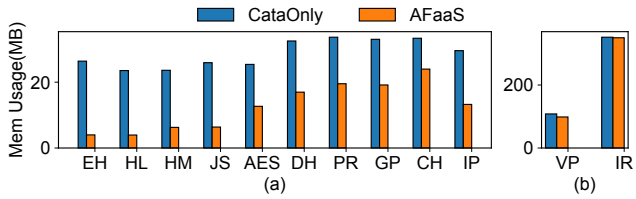


Figure 17: **Memory costs of level-2 seeds.**

6.6 Real-World Workload in Production

End-to-End Latency. AFaaS is widely deployed in Ant Group. To demonstrate the effectiveness of AFaaS, we compare the performance of AFaaS with CataOnly in production environment. We select eight representative Node.js functions for comparison (Table 2). For AFaaS, we perform a statistical analysis of the average end-to-end latency on the server for one day. For CataOnly, we test on the same hardware machine and follow the same pattern (mocked peer call responses).

As shown in Figure 18, the overall speedup ranges from 1.80× to 8.14×. Among them, the *encoder* achieves a lower speedup of 1.80× because its user code incurs higher execution latency. The *cc* function achieves a higher speedup of 8.14× due to reduced module load time. AFaaS maintains a stable startup latency from 5.45ms to 9.41ms.

Deployment Cost. Table 3 summarizes the memory sizes of seeds at different levels in production. Most functions achieve millisecond-level startup with a memory cost ranging from $6.9MB - 135.03MB$. This approach offers a better cost-effective alternative compared to hot start solutions that cache instances (for 1 minute).

Func	Description
mongo	Import an SDK and query the backend DB service.
bff	A service coordinates requests to multiple interfaces. [9].
cc	Schema-based code generator.
png	An image tool merges existing images to create new ones.
nunjucks	Render the template to HTML using Nunjucks [67].
mock	Generate random user data for unit testing.
encoder	Generate the base64 data for a QR code image.
compile	Input a JS file, and build the JS file into V8 code cache.

Table 2: **Representative functions in production.**

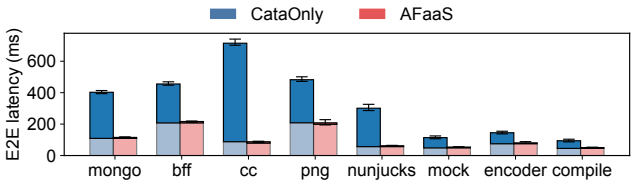


Figure 18: **End-to-end latency of the representative functions in production.** Each bar has two parts: the upper dark part shows the total time for control-path, container initialization, and user code initialization, while the lower light part indicates user code execution time.

Seeds	Memory	Seeds	Memory
level 0	14.31 MB		
level 1 (Node.js)	54.01 MB	*level 1* (Python)	29.05 MB
level 2 (mongo)	54.40 MB	*level 2 (bff)*	63.84 MB
level 2 (cc)	135.03 MB	*level 2 (png)*	90.20 MB
level 2 (nunjucks)	63.70 MB	*level 2 (mock)*	51.40 MB
level 2 (encoder)	9.46 MB	*level 2 (compile)*	6.90 MB

Table 3: **Memory cost of different level seeds in production**.

7 Discussion

Security Analysis. AFaaS follows the threat model of existing secure containers (e.g., Catalyzer [28], FireCracker [5], RunD [58], Kata Containers [31], gVisor [40]), where apps within the container could be compromised or malicious [37, 97, 98] and the sandbox and underlying host kernel are considered potential attack targets. Secure containers reduce the attack surface caused by traditional containers sharing the host kernel. Compared to gVisor, which both Catalyzer and AFaaS are based on, we make design choices to enhance performance. Below, we first examine the security implications of resource pooling and sharing strategies, followed by analyzing other security limitations in the current implementation.

(1) Security of Resource Pooling: (a) veth pool: AFaaS ensures that long-lived connections are gracefully terminated before reclaiming the device to prevent new users of the device from receiving unexpected data packets. (b) cgroup pool:

There is only one container inside the cgroup, and each cgroup is properly released after the exit of the function. Similar mechanisms have been used in previous work [38, 58].

(2) Security of Resource Sharing: (a) network namespaces: Network resources in AFaaS, such as devices, IP addresses and routing tables, are controlled within the guest OS. Therefore, sharing the host's network namespace between seed and sandbox does not pose any security issues. (b) IPC namespaces: The IPC namespace on the host is used mainly to isolate communication through shared memory. It is secure because there are no use cases for inter-sandbox (VM-process) IPC communication through shared memory in AFaaS.

AFaaS enables memory pages to be shared across different levels of seeds and instances, which may result in potential leakage of seeding data. However, instances can only access their own seed data, not that of others. This aligns with prior seed-based works (e.g., Catalyzer [28], Molecule [27], MITOSIS [96]) and does not introduce new side-channel attack risks. To enhance security, we audit function behavior, including IP addresses, ports, and protocols for ingress and egress.

Overall Impact. AFaaS achieves near hot-start performance with minimal additional resource consumption. Compared to strict cold starts, optimization techniques such as pooling, sharing, and seeding incur predictable overhead — a controlled trade-off inherent in production-grade serverless platforms. Beyond the memory footprint of the seed, AFaaS preallocates specific kernel objects, such as veths and cgroups, to alleviate contention bottlenecks. This allocation is limited and lightweight. Specifically, we pre-allocate 1,000 veths and 600 cgroups, resulting in negligible impact on the host OS. Moreover, the scheduler guarantees that function instances remain within the resource capacity of the node.

Generality. The design of AFaaS is broadly applicable to other platforms. First, AFaaS transforms the standard, general-purpose OCI into the specialized and efficient FRI that prioritizes performance over flexibility. Though this tight coupling with FaaS requirements necessitates runtime-specific adaptations, FRI's containerd-plugins architecture proves especially advantageous in performance-critical scenarios. Second, veth, cgroup, namespace, and seccomp are essential components of containers. Current container solutions also experience performance degradation under high stress. Our proposed solutions in this work are practical and deployable. Third, Google Cloud, AWS, and Alibaba Cloud could also benefit from our tree-structured seeds and multi-level fork design by integrating with VM fork [54].

8 Lessons Learned and Experiences

Workload Diversity. Serverless workloads exhibit behavioral divergence across cloud environments. In private clouds, Meta's XFaaS [75] emphasizes resource efficiency for latency-tolerant, non-interactive triggers (e.g., queues, events, and timers) through batched execution. In contrast, public clouds highlight diverse cold-start challenges: 80% of functions in Alibaba's FaasNet and 50% in Microsoft Azure run for over one second [78, 92], while AWS reports that 1% of invocations face cold starts lasting 100ms to 1s [8]. AFaaS operates under distinct constraints: short-lived functions and frequent cold starts demand millisecond-level initialization, a gap this work aims to address.

Fork's Renaissance in Serverless Systems. While prevailing approaches like caching [11, 99] and C/R [12, 19, 20, 56] achieve sub-second cold starts, they face fundamental limitations: caching relies on pre-warmed containers but struggles with unpredictable invocation patterns, while C/R incurs state restore overheads that cap latency reductions. Our production experience demonstrates that *fork*, when augmented with systematic optimizations, redefines viability for latency-critical workloads. Unlike probabilistic caching or I/O-bound C/R, *fork* enables **deterministic millisecond-level cold starts** through direct memory inheritance while preserving security via CoW semantics. This paper details how we overcome *fork*'s limitations (e.g., resource contention) through system optimizations, achieving production-scale deployment with many daily *forks* at millisecond-level (and even sub-millisecond) latency.

AFaaS Deployment Experience. While fork-based approaches offer advantages, there are still some issues that went wrong or were unexpected. We present insights that could be beneficial to know in advance for deployments.

(1) Special handling of uniqueness states is necessary to prevent unexpected behavior. While reusing a running seed's state can reduce startup latency, unique states such as random numbers or cryptographic tokens must be reinitialized carefully instead of being reused directly. For example, gVisor [40] uses a buffered I/O mechanism to batch *getrandom()* syscalls from the host kernel, reducing the frequency of host syscalls. However, this approach unintentionally duplicates random sequences in forked instances due to inherited states. To prevent security risks, AFaaS must ensure proper isolation and re-initialization of these states after fork operations. Similar issues exist in other systems, e.g., Firecracker explicitly warns against entropy reuse in restored microVM snapshots [29] due to the same risk.

(2) Bottlenecks under ultra-high concurrency and co-located deployment. Although AFaaS has optimized the main resource contention, some competition situations can still be observed in extremely high concurrency and co-located deployment scenarios [43]. First, the seed itself may become a bottleneck because only one instance can be cloned from it at a time (serially) as concurrency increases. AFaaS further enhances concurrency by deploying multiple instances of the same seed type. Second, when functions are repeatedly created and destroyed at a high frequency over an extended period, seccomp installation failures may arise due to bpf_jit_limit leaks [16]. Third, in co-located deployment

environments, other workloads (e.g., long-running microservices, large-scale data processing) on the node may acquire and hold the global lock of a cgroup. While AFaaS utilizes a cgroup pool to mitigate resource conflicts, the assignment or recycling of a cgroup to an instance still requires lightweight locks. This can potentially lead to performance jitter under high-concurrency scenarios.

(3) Don't be a noisy neighbor during preparation. AFaaS needs to prepare seed trees and resource pools, such as veth pairs and cgroups, in advance for function requests. Although the allocation is lightweight, the preparation process must avoid disrupting other workloads. First, not all kernel resources require pooling; it is crucial to identify the minimal necessary set. For example, resources like cpuset and cpuacct within cgroups do not need to be pooled. Second, creating resource pools (e.g., cgroup pools or veth pools) for AFaaS during system initialization can affect node workloads. For instance, the *netlink* operation when initializing the veth pool may lead to contention over the *cb_mutex*, resulting in spikes in both pod and node loads. To mitigate this, initialization should be performed serially. Lastly, if there are too many user-specific seeds, the system's memory pressure will increase, and AFaaS will swap some data to disk.

9 Related Work

Optimizations for Cold Start. Besides Catalyzer [28], other techniques also adopt *fork*-based methods [27, 68, 90, 96] to optimize cold start, but often overlook end-to-end life cycle gaps in production environments. These gaps are relevant to various cold start technologies, including lightweight virtualization [5, 31, 40, 58, 64], unikernels [50, 53, 71, 85], new container types [24, 52, 86], container C/R [13, 19, 28, 38, 51, 55, 56, 70, 79–81, 89, 94, 95], caching [17, 33, 63, 73, 74, 93, 99], provisioning [15, 60, 66, 92], and sharing [38, 48, 59, 76, 100, 101]. Notably, SOCK [68], SEUSS [19], and RainbowCake [99] share similarities with our tree-based hierarchical design and also incorporate user code into the seed. For example, RainbowCake [99] introduces a layer-wise container pre-warming and keep-alive strategy. However, unlike these systems, we prepare seeds exclusively during seed preparation (§4.3) and do not support dynamic seed scaling. Our approach is simpler and more practical, having been validated for effectiveness in industrial deployments.

Hot Function Optimizations. There are works that optimize hot functions for throughput and concurrency. Nightcore [42] focuses on efficient processing of internal function calls generated during the execution of a serverless function. MXFaaS [84] shares CPU, I/O, and memory between concurrent invocations of the same function. 95% of the requests on our platform are related to high-frequency functions with hot starts, which can be improved with these works. This paper addresses the remaining 5% infrequent requests (covering over 50% of functions) related to cold start issues.

Control Plane and Orchestration. Most FaaS orchestration efforts, beyond mainstream solutions like Kubernetes [14, 18], focus on improving scheduling policies [21, 32, 42, 57, 82, 87, 88] and load balancing [4, 34]. Studies like Dirigent [26] and [61] highlight control plane bottlenecks in current FaaS cluster management. Dirigent [26] introduces a container orchestration system designed specifically for FaaS, aiming to reduce communication overhead and maximize throughput. These approaches are orthogonal to AFaaS and can complement each other effectively.

Industry Perspective. Recently, platforms such as Azure Functions [78], Alibaba Cloud Functions [58, 92], AWS Lambda [17, 83], Meta [75], and Huawei Cloud [20, 44, 45] have reported their serverless practices, workload characteristics, and optimization strategies [77]. However, prior studies have not thoroughly analyzed cold start issues within complex production environments or addressed unresolved real-world gaps from an end-to-end perspective, which are the key contributions of this paper.

10 Conclusion

This paper addresses the cold start challenge in serverless computing, focusing on industrial-scale deployment. Through an in-depth analysis of existing solutions, we identify three key bottlenecks: (1) inefficient control-path interactions, (2) resource contention under concurrency, and (3) user code initialization overhead. To mitigate these issues, we propose AFaaS. AFaaS replaces the heavyweight OCI with an efficient FRI in the control path, pools and shares resources to alleviate contention, and uses tree-structured seeds with multi-level forking for fast forks and efficient memory use. Our evaluations show that AFaaS reduces the cold start latency by 1.8× to 8.14× compared to state-of-the-art solutions, achieves <15 milliseconds at 24x concurrency, and decreases memory usage by up to 84.9%. Deployed in Ant Group's production environment for over 18 months, AFaaS demonstrates robustness and cost-effectiveness.

Acknowledgments

We sincerely thank our shepherd Tianyin Xu and the anonymous reviewers, whose reviews and suggestions greatly strengthened our work. We also thank Jialiang Huang, Chao Dong, Mingzhi Zhou, Xiaoying Liu, and Lin Zhu for their valuable help and support. This work was supported by Quan Cheng Laboratory (QCLZD202305), National Science and Technology Major Project (2024ZD0803000), National Natural Science Foundation of China (NSFC) under grant 92467102 and national key laboratory under grant 231HFD0501. Correspondence to: Dong Du (Dd_nirvana@sjtu.edu.cn), Yu Chen (yuchen@tsinghua.edu.cn).

References

[1] Aws lambda execution environments. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html. Accessed: 2024-10-31.

[2] Min instances for cloud functions. https://cloud.google.com/functions/docs/configuring/min-instances. Accessed: 2024-10-31.

[3] Django. https://www.djangoproject.com/, Dec 2024. Accessed: 2024-12-02.

[4] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. EuroSys '23, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[6] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[7] Alibaba. Alibaba function compute. https://alibabacloud.com/product/function-compute, 2023. Accessed: 2024-11-20.

[8] Amazon Web Services. Understanding the lambda execution environment lifecycle. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html#cold-start-latency. Accessed: 2025-03-10.

[9] Amazon Web Services. Backends for Frontends Pattern. https://aws.amazon.com/cn/blogs/mobile/backends-for-frontends-pattern/, 2023. Accessed: 2024-06-20.

[10] Amazon Web Services. Aws lambda. https://aws.amazon.com/lambda, 2024. Accessed: 2024-11-20.

[11] Amazon Web Services. Configuring provisioned concurrency for a function. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html, 2024. Accessed: 2024-12-1.

[12] Amazon Web Services. Improving startup performance with Lambda SnapStart. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html, 2024. Accessed: 2024-11-20.

[13] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[14] The Kubernetes Authors. Kubernetes. https://kubernetes.io/, 2024. Accessed: 2024-11-20.

[15] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 153–167, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Daniel Borkmann. Adjust insufficient default bpf_jit_limit. https://github.com/torvalds/linux/commit/a4bbab27c4bf69486f5846d44134eb31c37e9b22, 2021. Accessed: 2025-05-28.

[17] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association.

[18] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, January 2016.

[19] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, Wei Liu, Linfeng Li, Fangming Liu, and

Kun Tan. Yuanrong: A production general-purpose serverless system for distributed applications in the cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 843–859, New York, NY, USA, 2024. Association for Computing Machinery.

[21] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 251–263, Santa Clara, CA, July 2017. USENIX Association.

[22] containerd. Containerd runtime v2. https://github.com/containerd/containerd/tree/main/core/runtime/v2, 2024. Accessed: 2024-12-1.

[23] Containerd contributors. Containerd: An industry-standard core container runtime. https://containerd.io/, 2024. Accessed: 2024-11-13.

[24] Marcin Copik, Alexandru Calotoiu, Gyorgy Rethy, Roman Böhringer, Rodrigo Bruno, and Torsten Hoefler. Process-as-a-service: Unifying elastic and stateful clouds with serverless processes. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 223–242, New York, NY, USA, 2024. Association for Computing Machinery.

[25] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. Dirigent: Lightweight serverless orchestration. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 369–384, New York, NY, USA, 2024. Association for Computing Machinery.

[27] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Feb 2022.

[28] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Firecracker Maintainers. Firecracker snapshotting: Randomness for clones, 2023. Accessed: 2025-05-23.

[30] The Linux Foundation. Open container initiative (oci). https://opencontainers.org/, 2023. Accessed: 2024-11-20.

[31] The OpenStack Foundation. *Kata Containers Architecture*. https://github.com/kata-containers/documentation/blob/master/design/architecture.md.

[32] Alexander Fuerst, Abdul Rehman, and Prateek Sharma. Ilúvatar: A fast control plane for serverless computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 267–280, New York, NY, USA, 2023. Association for Computing Machinery.

[33] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Alexander Fuerst and Prateek Sharma. Locality-aware load-balancing for serverless clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 227–239, New York, NY, USA, 2022. Association for Computing Machinery.

[35] GeeksforGeeks. Angularjs tutorial. https://www.geeksforgeeks.org/angularjs/, July 2024. Accessed: 2024-12-1.

[36] Google. Google serverless computing. https://cloud.google.com/serverless, 2022. Accessed: 2024-11-20.

[37] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: The bewildered eBPF on clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988, Anaheim, CA, August 2023. USENIX Association.

[38] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma,

Haifeng Gong, and YongWei Wu. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[39] Huawei. Huawei cloud functions. https://developer.huawei.com/consumer/en/agconnect/cloud-function/, 2022. Accessed: 2024-11-20.

[40] Google Inc. *What is gVisor?* https://gvisor.dev/docs/.

[41] Open Containers Initiative. Open container initiative runtime specification. https://github.com/opencontainers/runtime-spec/blob/main/runtime.md, 2024. Accessed: 2024-11-20.

[42] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. ASPLOS '21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cérin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing co-located workloads in alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 10(4):2381–2397, 2022.

[44] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 443–458, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless cold starts and where to find them. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 938–953, New York, NY, USA, 2025. Association for Computing Machinery.

[46] Kata Containers Contributors. What is vm templating and how do i use it. https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/what-is-vm-templating-and-how-do-I-use-it.md. Accessed: 2025-05-10.

[47] Benjamin Kettner and Frank Geisler. *Azure Functions*, pages 49–78. Jan 2022.

[48] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. JITServer: Disaggregated caching JIT compiler for the JVM in the cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 869–884, Carlsbad, CA, July 2022. USENIX Association.

[49] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, Nov 2019.

[50] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[51] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 298–316, New York, NY, USA, 2024. Association for Computing Machinery.

[52] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. Function as a function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 81–92, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

[55] Yuqiao Lan, Xiaohui Peng, and Yifan Wang. Snapipeline: Accelerating snapshot startup for faas containers. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page

144–159, New York, NY, USA, 2024. Association for Computing Machinery.

[56] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. Sabre: Hardware-Accelerated snapshot compression for serverless MicroVMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 1–18, Santa Clara, CA, July 2024. USENIX Association.

[57] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 32–47, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, Carlsbad, CA, July 2022. USENIX Association.

[59] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[60] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. Flashcube: Fast provisioning of serverless functions with streamlined container runtimes. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, PLOS '21, page 38–45, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The gap between serverless research and real-world systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 475–485, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Google LLC. Site reliability engineering: Service level objectives. https://sre.google/sre-book/service-level-objectives/, 2024. Accessed: 2024-01-17.

[63] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[64] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[65] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[66] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, page 21, USA, 2019. USENIX Association.

[67] Mozilla. Nunjucks. https://mozilla.github.io/nunjucks/. Accessed: 2024-12-10.

[68] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[69] Open Containers Initiative. About the open container initiative. https://opencontainers.org/about/overview/, 2023. Accessed: 2023-10-23.

[70] Xingguo Pang, Yanze Zhang, Liu Liu, Dazhao Cheng, Chengzhong Xu, and Xiaobo Zhou. Expeditious High-Concurrency MicroVM SnapStart in persistent memory with an augmented hypervisor. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 985–998, Santa Clara, CA, July 2024. USENIX Association.

[71] Federico Parola, Shixiong Qi, Anvaya B. Narappa, K. K. Ramakrishnan, and Fulvio Risso. Sure: Secure unikernels make serverless computing rapid and efficient. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 668–688, New

York, NY, USA, 2024. Association for Computing Machinery.

[72] Pivotal Software. Spring Framework. https://spring.io/projects/spring-framework, 2024. Accessed: 2024-11-27.

[73] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. ASPLOS '22, page 753–767, New York, NY, USA, 2022. Association for Computing Machinery.

[75] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 231–246, New York, NY, USA, 2023. Association for Computing Machinery.

[76] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Carlos Segarra, Ivan Durev, and Peter Pietzuch. Is it time to put cold starts in the deep freeze? In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 259–268, New York, NY, USA, 2024. Association for Computing Machinery.

[78] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[79] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[80] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: a fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[81] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[82] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[83] Josef Spillner. Quantitative analysis of cloud function evolution in the aws serverless application repository. 05 2019.

[84] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[85] Syswonder. Ruxos: A rust-based unikernel framework. https://github.com/syswonder/ruxos, 2023. Accessed: 2025-05-25.

[86] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery.

[87] Wenda Tang, Yutao Ke, Senbo Fu, Hongliang Jiang, Junjie Wu, Qian Peng, and Feng Gao. Demeter: Qos-aware cpu scheduling to reduce power consumption of multiple black-box workloads. SoCC '22, page 31–46, New York, NY, USA, 2022. Association for Computing Machinery.

[88] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 78–93, New York, NY, USA, 2022. Association for Computing Machinery.

[89] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[90] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 148–162, New York, NY, USA, 2005. Association for Computing Machinery.

[91] Gil Walker. wrk: A modern http benchmarking tool. https://github.com/wg/wrk, 2024. Accessed: 2024-04-05.

[92] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[93] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[94] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[95] Ziqi Wang, Chul-Hwan Choo, Michael A. Kozuch, Todd C. Mowry, Gennady Pekhimenko, Vivek Seshadri, and Dimitrios Skarlatos. Nvoverlay: Enabling efficient and scalable high-frequency snapshotting to nvm. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 498–511, 2021.

[96] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.

[97] Jietao Xiao, Nanzi Yang, Wenbo Shen, Jinku Li, Xin Guo, Zhiqiang Dong, Fei Xie, and Jianfeng Ma. Attacks are forwarded: Breaking the isolation of MicroVM-based containers through operation forwarding. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7517–7534, Anaheim, CA, August 2023. USENIX Association.

[98] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[99] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 335–350, New York, NY, USA, 2024. Association for Computing Machinery.

[100] Xinmin Zhang, Qiang He, Hao Fan, and Song Wu. Faascale: Scaling microvm vertically for serverless computing with memory elasticity. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 196–212, New York, NY, USA, 2024. Association for Computing Machinery.

[101] Yifei Zhang, Tianxiao Gu, Xiaolin Zheng, Lei Yu, Wei Kuai, and Sanhong Li. Towards a serverless java runtime. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 1156–1160. IEEE Press, 2022.

Kamino: Efficient VM Allocation at Scale with Latency-Driven Cache-Aware Scheduling

David Domingo* *Rutgers University*
Hugo Barbalho *Microsoft Research*
Marco Molinaro *Microsoft Research*
Kuan Liu *Microsoft Azure*
Abhisek Pan *Microsoft Azure*
David Dion *Microsoft Azure*
Thomas Moscibroda *Microsoft Azure*
Sudarsun Kannan *Rutgers University*
Ishai Menache *Microsoft Research*

Abstract

In virtual machine (VM) allocation systems, caching repetitive and similar VM allocation requests and associated resolution rules is crucial for reducing computational costs and meeting strict latency requirements. While modern allocation systems distribute requests among multiple allocator agents and use caching to improve performance, current schedulers often neglect the cache state and latency considerations when assigning each new request to an agent. Due to the high variance in costs of cache hits and misses and the associated processing overheads of updating the caches, simple load-balancing and cache-aware mechanisms result in high latencies. We introduce *Kamino,* a high-performance, latency-driven *and* cache-aware request scheduling system aimed at minimizing end-to-end latencies. Kamino employs a novel scheduling algorithm grounded in theory which uses partial indicators from the cache state to assign each new request to the agent with the lowest estimated latency. Evaluation of Kamino using a high-fidelity simulator on large-scale production workloads shows a 42% reduction in average request latencies. Our deployment of *Kamino* in the control plane of a large public cloud confirms these improvements, with a 33% decrease in cache miss rates and 17% reduction in memory usage.

1 Introduction

Large cloud providers invest billions of dollars in infrastructure to accommodate the growing demand for different forms of virtual machines (VMs) [16]. The VM allocation systems, which assign VM requests to the physical hardware, are therefore considered critical in the cloud stack. These systems are designed to satisfy two main goals: (i) complete the VM assignment within a stringent latency upper bound (in the order of tens of milliseconds); (ii) achieve "high quality" allocations, which accounts for a variety of provider and customer preferences; one example here is placing the VMs in a way that servers are highly utilized, which leads to high infrastructure Return of Investment (ROI) [16]. Unfortunately, these two design goals are inherently at odds as high-quality allocations require considering numerous objectives and constraints that must be evaluated over huge inventories (e.g., hundreds of thousands of servers). This typically makes the process computationally intensive [13, 31, 36], and a potential bottleneck at high-load scenarios [6, 16].

Cloud providers have dedicated substantial efforts to sustaining low VM allocation latencies without giving up on adequate quality. One standard approach is to utilize multiple allocation agents (or instances) over the same hardware inventory (e.g., an availability zone). A common design choice is to include multiple such allocator agents (henceforth referred to as AA) in a single node (a physical server) and further scale out by using multiple nodes [16, 36, 41]. Intra-node scaling of AAs exploits multicore parallelism and reduces the communication, consistency, and resource usage overheads that arise from scaling AAs across multiple nodes. This paper focuses on the mechanism for deciding which AA should be assigned to handle each incoming request.

In addition to having multiple AAs, a complementary approach to reduce latency and increase throughput when resolving allocation requests is to employ in-memory caching. State-of-the-art VM allocators, such as those employed in Azure, use a domain-specific caching layer [16, 41, 43]. However, unlike traditional caches that store only data objects, VM allocators store a combination of rules with fully or partially computed inventory state and a preferred set of servers for each allocation request type. Additionally, each cache entry size can vary from 10 to 100 MB, which enforces limits on the number of in-memory cache slots. A cache hit involves a combination of cache lookups and minor computations. In contrast, a cache miss might increase the latency to hundreds of milliseconds due to computational and data access costs, which vary significantly based on the complexity of allocation request types. Finally, some VM allocators use private AA caches to avoid concurrency control and synchronization overheads [16].

A key limitation of the state-of-the-art VM allocator de-

*Part of this work was done during an internship at Microsoft Research, Redmond

signs is the *lack of synergy between scheduling a request to an AA and the underlying caching mechanism.* More precisely, state-of-the-art VM allocators resort to standard load balancing techniques [16, 43], where requests are scheduled to an AA even if the request type does not reside in the AA's cache. The schedulers use simple strategies such as round-robin, work-stealing, uniform random assignment, or more complex hashing-based approaches to balance requests across the AA. Unfortunately, these strategies might lead to cache misses and computational overheads at production scale, with thousands of request types over a large inventory of up to hundreds of thousands servers, resulting in excess allocation latencies.

A different approach is to take inspiration from other domains, such as processor caches [26], content delivery networks (CDNs) [48], and storage caching [4], to design cache-aware request scheduling to the AAs. A plausible cache-aware scheduling policy would send requests of the same type to the same set of AAs. However, such simple cache-aware scheduling results in the following critical challenges: (1) Popular requests in certain workloads could create hot spots by scheduling requests to the same AA, increasing queuing delays and raising both average and tail latency. (2) A distinct challenge specific to VM allocators is that caches are generally hierarchical, with a smaller top-level cache that stores the information about the request type and a lower-level cache whose content may be consumed by different request types. Consequently, it is hard to a-priori estimate the resulting outcome of assigning a request to an AA; for example, partial hits are common, latencies are highly variable as a function of the request type, cache state, and system load, and different parts of that cache require more processing time to get up-to-date.

In view of the above challenges, we design and implement *Kamino* – a hybrid latency- and cache-aware VM allocation request scheduling framework for minimizing end-to-end request latency. At a high level, each AA is augmented with a request queue, and Kamino assigns each request to one of the queues based on latency considerations. The underlying algorithmic problem turns out to be a generalization of the classical problem of optimal online job scheduling for latency minimization [5], which is known to be notoriously hard in theory [5, 7]. Intuitively, our problem is even harder than the classical one because now the requests' processing times are not fixed and depend on which information is in the cache, which in turn depends on the scheduler's previous decisions. Nonetheless, the theoretical conceptualization allows us to design our scheduling algorithm based on first principles. We design a novel hybrid algorithm called *LatCache*, a latency-driven request scheduling algorithm that estimates latency even with access to only a set of partial indicators such as the type of requests in each queue and their cache hit predictions.

At its core, the key insight of the *LatCache* algorithm is to estimate the request latency within the target AAs' queues based on estimated processing time of the requests in each queue, and assign a new request to the queue where the end-to-end latency (queuing time + processing time) would be minimal. The processing time estimates are derived by an efficient analysis of the current cache state and how the queued requests effect the cache state as they are processed. This enables Kamino to dynamically adapt to any arrival pattern, while exploiting the AA caches (§2.2.1). While cache-aware load-balancing has been applied in previous works (see §7), a key novelty in our approach is considering the cache state, and in particular cache hit rates, as only one of the components contributing to end-to-end request latency, which is our actual metric of interest. Thus, we need to combine the state of different components (e.g., cache and queue) to efficiently estimate the request latency on the fly, also taking into account our hierarchical cache structure.

We implement and validate the effectiveness of Kamino and *LatCache*, first through high-fidelity simulations and then on Azure's production control plane. Kamino's architecture preserves a clear separation between the internal logic of VM assignments to physical machines and *LatCache* to isolate frequent modifications of the former. To sustain high throughput, we pipeline individual stages of Kamino, such as the request classification and queue assignment.

We compare Kamino against state-of-the-art techniques, including Protean, Azure's VM allocation system, and consistent hashing, which is widely used for workload balancing in other distributed systems [47]. Our simulation using multi-region production allocation traces indicate that Kamino obtains an average 42% reduction in latency over the latency agnostic scheduler. As a by product, *LatCache* accommodates up to 2x more throughput, which is especially significant in bursty load periods.

We deployed Kamino with a simple version of the *LatCache* algorithm in all production zones (with up to hundreds of thousands of machines each), reducing allocation latency by 11.9% at the 90th percentile and reducing cache misses by 33%. Additionally, the memory footprint per AA is reduced by up to 17%, enabling more AAs and other services per control-plane machine.

In more detail, the practical significance of these improvements is the following: the tail latency reduction ensures that resource provisioning remains reliable, especially for latency-sensitive workloads like data analytics platforms, online gaming, and virtual desktops. High tail latency in VM allocation can delay autoscaling, causing unpredictable instance availability and degrading application performance. The improved cache miss rate reduces the number of machines required for AA, which decreases contention and communication overheads, and in turn conflicts and retries in the allocation system. This results in more efficient request handling and overall system stability. The AAs run in ringfenced control plane nodes instead of the general cloud fleet for better predictability and security, and share resources with hundreds of other control-plane microservices. Thus, having resource efficiency solutions, specifically memory efficiency, is critical.

The combined improvements in latency, throughput and resource consumption are enabling Azure to consistently serve hundreds of thousands VM requests in a zone, while freeing up compute for a growing number of other control-plane services.

Beyond VM allocators, we believe that our approach can extend to systems with variable request latencies, including Log-Structured Merge Trees (LSMs), distributed databases, content delivery networks (CDNs), and microservices architectures. Indeed, we conducted preliminary experiments employing latency and cache-centric mechanisms in an LSM-based key-value store. By accounting for the differing request latencies and locality awareness across multiple levels (as in Kamino), our results show a reduction in the overall request latencies (§A.1).

In summary, we make the following contributions:

- We provide a detailed analysis of the challenges in handling VM allocation requests at scale, zooming-in on issues related to latency and resource consumption.
- We propose novel latency- and cache-aware (*LatCache*) request scheduling algorithms that are theoretically sound and are based on careful estimation of latencies based on cache and queue state.
- We realize the algorithms by designing and implementing the Kamino framework. We first develop a high-fidelity request scheduling simulator and subsequently deploy the scheduling framework in production zones.
- We evaluate the benefits and implications of Kamino through an exhaustive simulation study of various plausible algorithmic approaches, followed by measurements from our large-scale production zones.

2 Background and Motivation

We provide a background on VM allocation followed by challenges on reducing VM allocation request latency, resource inefficiencies, and scaling the number of AAs.

2.1 Background on VM allocation systems

Designing low-latency VM (or container) allocators is critical for reducing the instantiation time of application services. It is crucial to do so while also satisfying the large variety of VM/container deployment requirements. State-of-the-art VM allocators like Protean and resource managers, such as Omega [36], Kubernetes [27], Twine [41], and VMWare DRS [15], achieve this through the use of filter *predicates* or *policies* that encapsulate allocation constraints to narrow down the available inventory into a set of candidate machines for a given request. Subsequently, these systems implement designated *preferences* and *priorities* to sort these candidate machines, ultimately establishing a preferred pool from which a machine can be selected.

Computationally-intensive rule-based allocations. VM allocators, like Protean, utilize *rule-based allocations*, where each *rule* defines the logic for filtering and ranking available inventory based on specified allocation constraints or preferences. These constraints and preferences can either be user-defined, such as VM type, or internally defined to enhance allocation strategies. Rules can be added to accommodate additional constraints or preferences as platform offerings grow and allocation strategies evolve.

Time	Request	Cached objects	Hits	Misses
1	$req(a_1,b_1)$	None		$cons(a_1,b_1)$ $R_1(a_1,b_1)$ $R_2(a_1)$
2	$req(a_1,b_2)$	$cons(a_1,b_1)$ $R_1(a_1,b_1)$ $R_2(a_1)$	$R_2(a_1)$	$cons(a_1,b_2)$ $R_1(a_1,b_2)$
3	$req(a_1,b_1)$	$cons(a_1,b_1)$ $cons(a_1,b_2)$ $R_1(a_1,b_1)$ $R_2(a_1)$ $R_1(a_1,b_2)$	$cons(a_1,b_1)$	

Table 1: Illustration of the hierarchical cache for a system with 2 rules, R_1, R_2, where the latter depends on only one the a-attribute of the request. Each request has 2 attributes, a and b. $cons(\cdot)$ denotes the consolidated node list for a given request. When a request comes, both its consolidated list and rule evaluation for R_1 and R_2 are brought into the cache, if not already present.

Evaluating and fulfilling an allocation request and its constraints is computationally intensive, involving the following steps: First, relevant rules are selected and sorted by priority, with constraints taking precedence. Then, each rule is applied to the inventory, yielding machine sets that satisfy specific constraints. The intersection of these sets is calculated and ordered using preference rules, resulting in a final candidate set that is ranked by both quality and preference, from which one of the top-ranked machine(s) is selected.

The benefits of hierarchical caching. To speed up allocation and reduce computation costs, state-of-the-art cluster managers use caching techniques [16, 36, 41, 43]. The cache design is centered around the observation that requests submitted close in time exhibit locality regarding constraints. State-of-the-art VM allocators like Protean use cache *slots* (logical cache entries) for minimizing re-computation. Specifically, Protean uses *hierarchical caching,* with two levels of caching: a top-level cache (fast path), which is used for resolving identical allocation requests, and a lower-level cache (slow path), used for requests that are non-identical but have a subset of identical traits or constraints (e.g., requests for the same VM size, but with different priority); see Table 1 for an illustration, and §4.2.3 for details. A fast path cache hit can have a significantly lower rule computation cost than a slow path hit. Ultimately, *the fast path reduces the hit latency of identical requests, whereas the slow path reduces miss latency for non-identical requests.* Finally, each cache slot could have a variable memory size, typically ranging from 10 to 100s of megabytes, depending on the complexity of the rules. This limits the number of slots reserved for an AA.

Cluster-level and node-level scaling of AAs. To operate at a large scale and serve thousands of allocation requests per second, scaling the number of allocators is critical. Scaling increases throughput and, importantly, can reduce request la-

tency by increasing cache hits and reducing time spent waiting for requests to be served. Protean employs two levels of scaling: (1) deploying multiple allocation agents (*AAs*) on nodes distributed across multiple clusters; (2) using multiple AAs within a single node to exploit multi-core parallelism. In the case of multi-node AAs, a frontend load-balancer dispatches requests to a specific node with one or more AAs. Within a node, multiple AAs form a pool, and agents in the pool pull and serve requests from the head of a global queue when they become free.

The distributed AAs operate concurrently and optimistically without frequent synchronization. However, concurrency may lead to conflicts in placement decisions. Hence, after evaluating requests, AA commits assignments as transactions to a global inventory store, establishing the inventory state. Conflicts are detected in the inventory store, prompting failed requests to be returned to AAs for rapid reevaluation. Successful allocations are committed, and changes are transmitted to an inventory pub/sub service, updating all AAs.

Despite these steps, *distributing the AAs across multiple nodes increases communication costs, conflict resolution costs, and under-utilization of per-node system resources.* Therefore, increasing the number of AAs within a single system is critical to exploit resources such as multi-core CPUs and reduce the communication and synchronization cost incurred by distributed AAs. The number of AAs in a node is dependent on the core count and memory available for caching besides using them for other management services.

Challenges of shared caches in scaling AAs. When scaling AAs, managing caches becomes a critical challenge. Sharing caches across multiple AAs can increase capacity and reduce redundancy but introduces significant drawbacks. First, shared caches incur heavy locking and synchronization overheads, limiting multi-core CPU efficiency and negating scalability benefits. These challenges are amplified in hierarchical caching, where distinct operations for fast and slow levels with substantially different cache hit and miss costs increase contention. Third, shared caches require periodic synchronization with a shared database, adding latency and communication costs. Compared to private caches, these issues make shared caches less efficient and more prone to stalling under high throughput, ultimately hindering scalability and performance. To our knowledge, state-of-the-art systems do not share caches across AAs. For example, Protean divides cache memory into slots and partitions it across AAs for maximum concurrency. However, reducing the number of per-AA cache slots to increase AA counts can lead to higher cache misses and poor request scaling (see §2.2).

Request assignment and scheduling algorithms. Request scheduling strategies play a crucial role in allocating incoming requests to AAs to achieve objectives such as latency reduction, throughput maximization, or resource utilization efficiency. While large-scale cloud providers often lack documentation on VM allocation request scheduling methods, systems like Protean utilize cache-oblivious scheduling policies like Round-Robin. Although effective in distributing requests with simplicity and generality, this approach's lack of cache awareness may lead to suboptimal performance. Similar strategies are employed in widely deployed systems like Kubernetes for container POD allocator scheduling [27, 29].

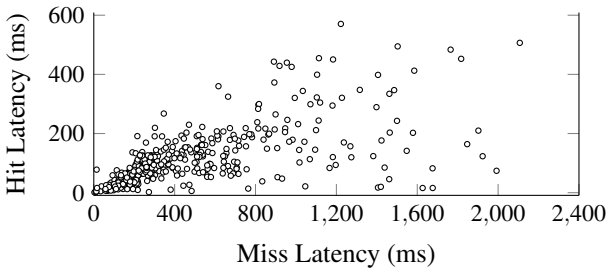


Figure 1: Scatter plot of avg. hit/miss latencies of 400 popular request types.

Cache-aware scheduling policies can leverage information about the AAs' caches and request types to assign each request type to a specific AA, maximizing cache hits and reducing request latencies. Prior works [9, 25, 47] on various other domains, including processor caches, CDNs, web caches, and distributed data caches, optimize performance by relying on cache affinity and utilizing dynamic mechanisms to handle potential load imbalances and adapting to workload changes. However, for VM allocation such strategies might lead to significantly higher latencies (see §6.2). Instead, our design primarily takes a latency-centric approach incorporating cache awareness, utilizing latency estimation to induce lower latencies. Intuitively, our approach can be viewed as 'load-balancing' in terms of the total 'work' done by each allocation agent (rather than traditional load-balancing, e.g., based on queue sizes or request types).

2.2 Evidence from production

To understand the challenges and limitations in state-of-the-art VM allocators, we characterize the latency performance of Protean, followed by assessing resource inefficiencies. We use traces from over 50 allocator nodes for our analysis, each containing a day's worth of requests.

2.2.1 Need for latency-driven scheduling

Our comprehensive study of latency characteristics in large-scale production traces emphasizes the necessity for a *latency-driven scheduling* framework.

Observation 1: Cache hit and miss latencies vary substantially across heterogeneous request types: In contrast to traditional data caching systems, the latencies for hit-and-miss events vary significantly based on the complexity of VM allocation queries among different request types. Figure 1 illustrates the distribution of hit/miss latencies observed by a single allocator node (hit and miss here refer to the top level of the cache). Some request types incur higher cache-hit latency than miss latency for others, contradicting typical assumptions

Node	Avg. Memory Usage	Avg. Peak CPU Usage
Node 1	∼92%	∼50%
Node 2	∼92%	∼61%
Node 3	∼90%	∼74%
Node 4	∼90%	∼39%

Table 2: Approximate resource usage within 4 nodes running allocation agents in a large availability zone.

that a cache hit is always faster than a miss. This variability arises from two primary reasons. Firstly, as discussed earlier, caches are hierarchical, and a request's constraints and preference rules may necessitate partial recomputation upon a cache hit or full recomputation in the case of a miss. Our analysis reveals up to a 5x variation in hit-and-miss latency across request types. Secondly, bursts in allocation requests stall and amplify the time it takes to resolve a request due to inventory updates, regardless of a cache hit or miss, further adding to variable request latency.

2.2.2 Need for cache-aware, latency-driven scheduling

Cache-awareness is critical for exploiting locality by placing similar requests on the same AAs. Consider a scenario with two request types (type 1 and type 2) and two AAs, each with a single cache slot for a specific request type, and a cache policy that adds to the cache the last request type processed by the AA. While cache-oblivious scheduling algorithms like Round-Robin could result in 0% cache hit rate (e.g., on the sequence with request types $1,2,2,1,1,2,2,\ldots$), the cache-aware scheduling that pins a request type to a specific AA could achieve a 100% cache hit rate, minimal queue lengths, and low total latency.

Indeed, Figure 6 in §6.2 shows that simple strategies are not effective enough: Round-Robin and random assignments have respectively $\approx 30\%$ and $\approx 60\%$ higher tail latency than Protean, which is essentially a pure work-stealing strategy.

Observation 2: Latency-driven cache-aware scheduling is critical for handling dynamic workloads: Unfortunately, just a cache-aware strategy like request-pinning or consistent-hashing is not sufficient. As discussed in §2.2.1, under higher load, with only a cache-aware approach, bursts of certain requests could significantly affect the request-type distributions across AAs. In contrast, an effective *latency-driven* scheduler would *dynamically* adapt to changes in latency and, if needed, spread out requests to other AAs even if it impacts locality.

2.2.3 Allocator scalability challenges

Increasing the number of AAs can reduce request wait times and latencies by using multicore CPU and available memory while ensuring sufficient throughput. However, scaling AAs across nodes or even within a node can lead to resource inefficiencies or increased latencies.

Observation 3: Resource imbalance between memory and CPU use limits the number of AAs: Increasing the number of AAs to reduce latency and improve throughput poses a challenge of resource utilization imbalance. The majority of

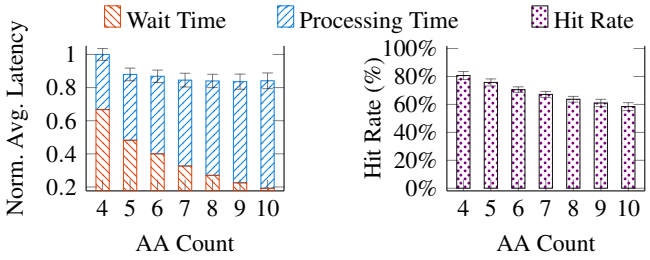


Figure 2: Performance as the number of AAs increases.

nodes exhibit high memory usage while under-utilizing CPU resources, as shown in Table 2. This imbalance arises from two main factors: first, hierarchical request evaluation caches require significant memory, with top-level slots encompassing the entire feasible inventory for a specific request type and lower-level slots containing feasible sets for each constraint. Second, as discussed above, to maximize parallelism and avoid synchronization bottlenecks, each AA maintains its private cache instead of using a shared cache. Due to these factors, our analysis system uses a maximum of four AAs per node to avoid exceeding the available memory limit, even for the largest zones. While scaling AAs across additional nodes is a potential solution, it would exacerbate CPU waste and communication and synchronization overheads.

Observation 4: Naively increasing the number of AAs by dividing cache slots affects latency: An alternative approach to increasing the number of AAs within a node is to divide the total available memory for caching across them further. However, reducing individual cache sizes can lead to more misses and, consequently, higher latency. To illustrate this, we simulate Protean and compare its performance with different AA counts using a day's worth of requests. We start with the baseline configuration of 4 AAs, increasing the AA count and keeping the cache size constant by reducing the size of each AA's private cache. We measure hit rate and latency and present the results in Figure 2. Increasing the AA count to 6-8 reduces request wait times in the queue, improving average latencies compared to the baseline (4 AAs). However, further increases in AA counts lead to smaller cache sizes, resulting in degraded hit rates and increased processing times due to a lack of cache-aware scheduling, where AAs indiscriminately pull requests off the node queue, failing to exploit locality across the AAs' caches.

The aforementioned *four observations drive the design of Kamino's latency-driven, cache-aware algorithm and supporting system implementation.*

3 Kamino overview

We next provide a high-level overview of our VM allocation system, focusing on the allocation path. In §5, we describe the details of the actual architecture.

System overview. Kamino is designed to minimize VM request latencies through concurrency and latency- and cache-aware scheduling; see Fig. 3. Each allocation node employs

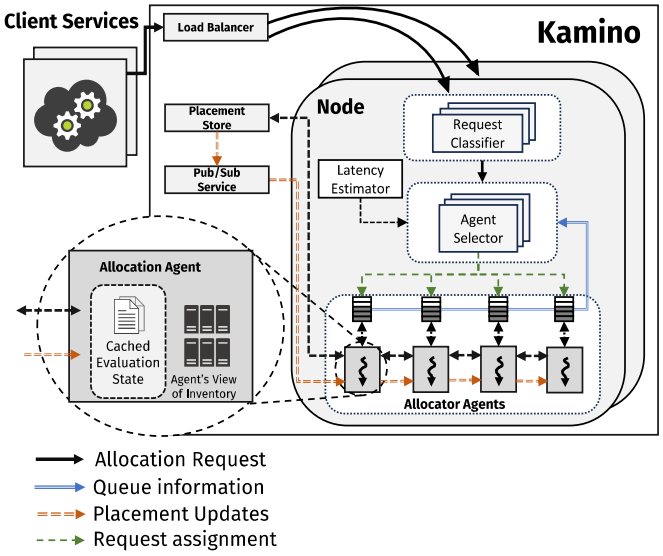


Figure 3: System Architecture with Kamino

multiple concurrent AAs tasked with evaluating rules and assigning VM requests to machines. AAs utilize hierarchical private caches for the evaluations and maintain queues for pending requests. Additionally, AAs have access to real-time data about the machines in the inventory, including available resources like CPU cores and memory.

Request allocation path. When a request enters the system, a gateway/load-balancer routes it to an allocator node. Within the node, the request's estimated latency is computed based on historical latencies as well as the current queue and cache states of each AA. The request is assigned to the AA that is estimated to complete processing the request the earliest. The selected agent runs the rules on the traits of the request, using its hierarchical cache where possible. This results in a list of machines that can receive the VM request, sorted by preference (according to the rules). The VM request is sent to one of the top-ranked machines of this list. The placement decision is committed to a central placement store, and the update is propagated across AAs.

4 *LatCache* Agent Assignment Algorithm

We now zoom in on a central part of our work: directing the incoming VM requests to the AAs in a way that reduces average and tail request latency by best utilizing their memory (i.e., cache) and computing resources. Since each allocator node is treated independently, we focus on a single node and on selecting which of its AAs agents should handle a given VM request, i.e., the design of the *agent assigner algorithm*.

4.1 Agent Assignment Task

We introduce the following task that captures the main elements of the situation at hand and is used as a guide in the design of our assignment algorithm.

An instance of the task consists of multiple AAs, and a sequence of heterogeneous incoming requests to be processed by the AAs. The AAs are identical and run in parallel. Each AA can only process one request at a time, and no preemptions are allowed. Requests cannot be dropped, and each AA has a queue with enough space to hold its pending requests; queued requests are processed in FIFO order.

Each AA has a private cache of bounded size that influences the processing time of a request. At a high level, the current state of the cache affects the processing time in a non-trivial way: it partially reduces the processing time, the extent of which depends on how much of the request is cached (i.e., which of its rule evaluations are cached, see §2.1), and the same portion of the cache (i.e., rules) may help different requests. In addition, even with the same cache state, the processing time of a request may change over time. This is because evaluating a cached rule depends on the changes to the inventory since the last update, which depends on the load of the system. Here, we abstract some of these details and refer the reader to §4.2.3, where we provide lower-level details of our hierarchical cache and how exactly its state affects processing time. Each AA also runs an independent caching policy that decides what data is evicted when the cache is full. Typically, after a request is processed by an AA, it is brought into the AA's cache, but this is not a required assumption.

When a new request arrives, it needs to be immediately assigned to one of the AAs. The goal is to assign the incoming requests to AAs so as to minimize the average latency, i.e., completion time (which includes waiting time) minus release time, of the requests.

Figure 4 provides an illustration of the agent assignment task. In this example, there are 2 AAs, each having a simplified single-slot cache, and two request types (green and orange). The left figure shows the state of the AAs' caches, queues, and current execution of AA 2 at time t_1, when a new request arrives, it needs to be assigned to one of the AAs; in this case, it is assigned to AA 1. The middle figure shows the state at time t_2, when AA 2 finishes its first request and will start processing the one in its queue; in this example, the cache of AA 2 was updated upon completion of the first request. The right figure shows the state at a later time t_3. Notice that the processing time of the second request of AA 2 is smaller, since its type was present in the cache when it started being processed (time t_2). Also, notice that the cache of AA 1 was updated due to the completion of the green request.

4.2 Latency-driven Cache-aware Assignment

4.2.1 Basic algorithm

We now describe *LatCache*, our proposed algorithm for this agent assignment task. Being latency-driven, the main component for making its decision is a careful estimation of what would be the latency incurred by the incoming request if it were assigned to a given AA.

To perform this estimation, we first break the latency into 3 separate components: *processing time*, *queue time*, and the *remaining processing time*. *Processing time* is the estimated

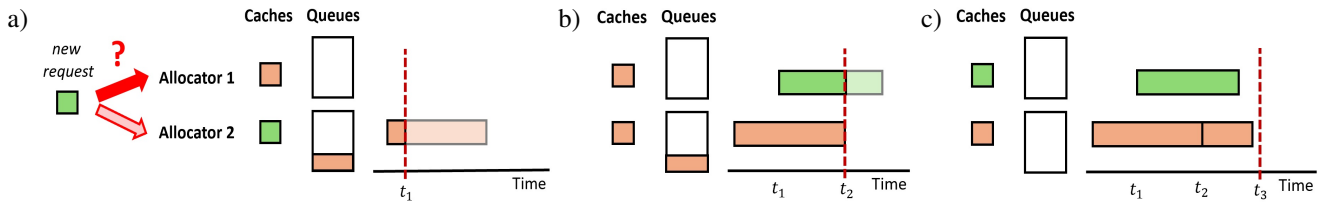


Figure 4: Illustration of the agent assignment task.

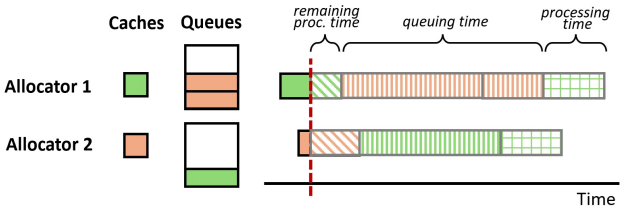


Figure 5: Illustration of *LatCache*'s latency estimation. A new green request (marked by a green square pattern) is considered to be placed on one of the two AAs, each with a single cache slot. Non-solid colors represent estimations.

time it would take to process the incoming request on a given AA, once it starts processing. *Queue time* is the estimated time for processing all the requests in the queue of the AA. *Remaining processing time* is the time left for the AA to finish the request currently being processed. The estimated latency of the request on an AA is obtained by adding the estimates for these three components.

Notice that all these 3 component depend non-trivially on the cache dynamics of the AA under consideration. For example, the *processing time* of the request does not depend on whether it is *currently* found (either fully or partially) in the cache of the AA, but whether the request would be there *by the time it would start processing*. This, in turn, depends on the current cache of the AA, the requests in its queue, and the caching policy it uses. Also note that a similar effect happens with the *queue time* of an AA, which depends on the processing times of the requests in the queue, which in turn depend on the extent that each request's information is present in the cache when processed. Thus, a crucial step in the algorithm is to obtain an adequate cache-aware estimate of each of the 3 latency components. We defer the description of how we estimate these components to §4.2.2.

Once the estimation of the latency of the request on each of the AAs is performed, *LatCache* simply assigns the request to the AA with the lowest estimated latency; see Algorithm 1 for the pseudo-code.

Algorithm 1 *LatCache* (*req* : request)

1: Compute cache-aware estimates $queueTime(a)$, $processingTime(a)$, and $remainingProcTime(a)$ for each allocation agent a
2: Assign *req* to the allocation agent a^* where it has minimum estimated latency $queueTime(a^*) + processingTime(a^*) + remainingProcTime(a^*)$

The design of *LatCache* addresses the challenges highlighted in the previous section. First, being latency-driven, it performs an adaptive load-balancing of requests to AAs, regardless of the identity of incoming requests (contrasting with the request-pinning strategy discussed in §2.2.2). Indeed, we establish that our algorithm maintains the queue waiting times of the AAs almost perfectly balanced; intuitively, if the queue of an AA grows too long, this is factored into the latency estimation of the algorithm which then chooses another AA for future requests.

Theorem 1 (*LatCache*'s load-balancing)**.** *Consider the algorithm LatCache with perfect latency estimation. Then, at all times, it guarantees that for every pair of AAs, their queue waiting times are within maxProcTime of each other, where maxProcTime is the largest request processing time.*

Moreover, no scheduler can guarantee that for every pair of AAs a, a' the difference in their queue waiting times is strictly less than maxProcTime at all times.

We note that the assumption of perfect latency estimation can be relaxed, and was made to simplify the bounds. In addition, with adequate cache-aware latency estimates *LatCache* also promotes co-location of similar requests, which should lead to higher cache hit rates and thus lower processing times: By construction, *LatCache* only prefers an AA that does not have (or has less of) the request in its cache over an AA that has the request in the cache if the latter has worse latency, e.g., due to significantly longer queuing time.

4.2.2 Cache-aware latency estimation

This desired behavior of *LatCache* is contingent on the algorithm computing adequate estimates for the three latency components, which we now address. Following up on previous discussion, the two key challenges here are: (1) It is not obvious whether the current request and the queued requests *will be* (fully or partially) found on an AA's cache, given that the state of the cache can change by the time each request is processed; (2) Estimation errors on the hit and miss latencies might accumulate (e.g., in the estimated queuing times) and lead to highly-suboptimal decisions. With this in mind, the latency components are estimated as follows:

Processing time. This is the main component to be estimated and is used to compute the other components. To estimate the $processingTime(a)$ of the incoming request on AA a, we employ *optimism* and assume that when this request is processed on a (if assigned to it), the cache will still contain all of its current content (i.e., none of it is evicted) *and* will

also completely contain all the requests that are currently in the queue of a (we call this the "augmented cache state"). The estimated $processingTime(a)$ is then a function of this "augmented cache state" and the traits of the current request (exact details of how this is computed in our situation are presented in §4.2.3, Algorithm 2).

We note that this optimistic cache state prediction is quite accurate as long as the number of distinct type of requests in the AAs' queues is small compared to how many types a cache can accommodate (which is what we see in practice for VM allocations). In particular, we tested an alternative estimation procedure that simulates future cache states and did not find significant improvement in estimation quality.

Queuing time. Next, the $queueTime(a)$ of an AA is obtained by estimating the processing time for each request in its queue, using the strategy outlined above. This is done without the need to traverse the queues: when a request is added to the queue of the AA a, the estimated *processingTime* of the request is added to the current $queueTime(a)$, and when a request is dequeued its estimated *processingTime* is subtracted. Note that by subtracting the *estimated* processing time on dequeue, as opposed to the real processing time, limits the accumulation of errors: once a request is dequeued, the effect that any estimation error on its processing time had on the queuing time is erased.

Remaining processing time. Finally, the estimation of the $remainingProcTime(a)$ of an AA is based on *start* and *proc_time* information that is updated by the AA as follows: whenever it starts processing a request, it sets *start* to be the current time, and *proc_time* to be the estimated processing time based on the current state of the cache (not the "augmented cache state"). With this information, $remainingProcTime(a)$ at time t is estimated as $max(0, start + proc_time - t)$ if the AA a is currently busy, else it is estimated as 0. Figure 5 illustrates these estimations.

4.2.3 Hierarchical cache structure and processing time computation

We next discuss rule-based allocations and the cache structure, essential for computing estimated processing times in Kamino.

Rule-based allocations. Our system utilizes *rule-based allocations*, as discussed in §2.1. A rule evaluated over a request returns a filtered (sorted) list of the current inventory based on specified constraints or preferences (e.g., the list of machines with enough free memory to accommodate the request). Each rule only depends on a subset of the traits of the request; requests with the same value on these traits produce the same evaluation, and thus are considered equivalent with respect to this rule. To fulfill a request, all rules are evaluated and lists consolidated, and the request is placed on one of the top-ranked machine(s) from the final list. These steps are performed by the AA handling the request.

Cache structure. The cache present in the AAs is used to speed up these steps. We employ a 2-level hierarchical cache: The top level (fast path) stores the consolidated list of machines for (equivalent) requests, and the lower level (slow path) stores the result of rule evaluations for equivalent requests (with respect to the rule). The latter allows for "partial cache hits" that benefit more requests since more requests are equivalent relative to a single rule than relative to the consolidated evaluation. See Table 1 for an illustration.

Since rule evaluations and their consolidated result depend on the current state of the inventory, they need to be updated even when present in cache; that is, both cache *hits* and *misses* add to the latency of processing the request. An un-cached rule evaluation (rule-miss) is computed from scratch, while cached rule evaluations (rule-hit) are updated with only the changes in the inventory change since its last update; this incremental update is significantly faster than computing it from scratch. If the consolidated evaluation for a given request is present in the top-level cache (hit), then all of the rules it depends on are updated, and the updates are inserted to/removed from the consolidated result. We note that while a consolidated result is present in the top-level cache, all rules that it depends on are pinned to the lower-level cache, i.e., all are rule-hits.

Processing time computation. Given this cache structure, Algorithm 2 describes our estimation of the processing time of a request *req* on a given AA. This procedure receives as input the "augmented cache state" *augCache* of the AA, as defined in §4.2.2, estimate *hit* for a cache top-level hit processing time, and vectors *rule-hits* and *rule-misses* with estimates for the rule-hit and rule-miss times of each rule. These input estimates can be computed, for example, using historical data, see §5. If the request is a top-level hit in the *augCache*, its estimated latency is only *hit*. In case of a top-level miss, we consider the "partial hit" based on the rules present in the lower level of *augCache*.

Algorithm 2 ProcessingTime (*req*, *augCache*, *hit*, *rule-hits* and *rule-misses*)

1: **If** the *req* is a top-level hit in *augCache*, then set *procTime* to be equal *hit*
2: **Else** #*top-level miss*
3: For each rule r, add to *procTime* either its hit time *rule-hits*$[r]$ or miss time *rule-misses*$[r]$ depending whether *augCache* contains the evaluation of R for request (equivalent to) *req*
4: **Return** *procTime*

5 System Implementation

We describe the details of the Kamino request scheduling framework implemented within the Protean system and highlight some practical challenges of the implementation.

Overall system organization. A logical instance of the Protean system is responsible for handling all requests in a zone or a region. The system consists of a pool of stateless AAs backed by a persisted placement store for VM placement

decisions. Each AA uses a dedicated thread and private in-memory state and caches to place VM requests to appropriate servers in the inventory. It uses a specified number of cache slots to limit its memory usage. Multiple AAs are grouped within a process in an allocator node, and multiple nodes can be employed to satisfy peak system demand.

The AAs follow an optimistic concurrency model: an agent makes placement decisions based on its (potentially stale) view of the inventory and persists each placement result to the placement store. The placement store rejects results that are invalid because of the staleness of the AA's view so that the requests can be retried. VM deletes are also persisted in the placement store. The AAs learn about updates to the placement store and other relevant inventory updates, such as those related to server health and power consumption, through a publish-subscribe platform. As an optimization, placement decisions made by the AAs within an allocator node are shared among each other through fast in-memory transfer. Figure 3 shows the overall system architecture.

Kamino request scheduling framework. The front-end gateway/load-balancer service accepts the requests and routes them to an allocator node, but Kamino is responsible for mapping the requests to AAs within a node. Kamino aims to minimize overall request latency and improve the resource efficiency of the AAs while incurring negligible computation and memory overheads. To that end, Kamino is implemented as part of the process the AA itself runs on, thus avoiding any inter-process communication overheads. In addition, we designed our scheduling logic in a computationally efficient manner, resulting in minimal overhead in the order of microseconds per request, which is orders of magnitude lower than request latencies (tens to hundreds ms); thus, the scheduler has negligible overhead on the total latency. Additionally, slower time-scale computations required for making scheduling decisions are done outside the request-handling critical path and so do not introduce any relevant computational overhead. Kamino consists of the *Request classifier*, *Agent selector*, and *Latency estimator* modules. The *Request classifier* and *Agent selector* processes each request in the critical path, whereas the *Latency estimator* runs outside the critical path. We describe these modules below.

Request classifier. A request arriving at the node remains queued until an instance of the request classifier becomes available to handle it. The request classifier computes the equivalence class key of the request depending on the rules relevant to the request and the request traits they depend on (see §2.1). The key uniquely identifies a class of requests that are equivalent to each other as far as the allocation logic is concerned. In other words, each key denotes a *request type* and determines if an AA has a cached consolidated result for the type. It decorates the request metadata with the computed key and queues the request for further processing. The framework allows for multiple instances of the request classifier (and agent selector) modules in order to concurrently process the requests, but in practice just one or two of these are enough because the processing latency of these modules is negligible compared to that of the AA.

Agent selector. Each AA has its own private queue to hold pending requests. The agent selector assigns each request to an AA's queue. Once a request is assigned to a queue, the assignment is never changed. This module implements the core request scheduling algorithm *LatCache*. As described in §4.2, this algorithm requires a few inputs: (1) It needs estimates of top-level cache hit times and hit and miss times for the rules. This is obtained from the *Latency estimator* module. (2) It needs to check whether an incoming request type is cached in an AA or is already queued in its private queue. The agent selector probes the AA's caches with the request type's key to check if it is cached. As part of the queue metadata for each agent, it maintains a map of <Request type key, Count> for the requests in the queue. This map is used to check if a request type is already present in the queue and is updated when requests are enqueued or dequeued. (3) It needs to check if an agent is busy. The agent selector holds a reference to the last dequeued work item for each agent and probes the work item to check if the agent is busy.

Latency estimator. This module is responsible for estimating the hit and miss times. In its simplest form, it can provide estimated hit or miss times from preset service configurations. However, a more practical implementation takes the form of a background task on each allocator node that tracks hit and miss times and periodically produces an updated estimate of their averages. The latest estimates are not only sent to the *Agent selector* but are also persisted in order to have a good initial estimate on process restart. Although we only use a rough estimate of central tendency for hit and miss times, *LatCache* is able to make good assignments and yield a significant improvement in the relevant metrics (see §6.4).

Cache management policy. The AAs employ a hybrid cache eviction policy – objects are evicted from the cache if it is full, using a standard LRU replacement policy, or if they reach a certain age. Age-based eviction allows us to reduce the agent memory footprint during periods of low load.

6 Evaluation

We conduct extensive experiments to evaluate Kamino using both a high-fidelity simulator and the production system. The simulator experiments utilize *real traces* from production across various geographical regions. Our experiments aim to address the following questions:

- Does Kamino and its latency-driven *LatCache* algorithm reduce VM allocation request latency over standard, well-known algorithmic baselines?
- How effective is *LatCache* in reducing cache misses and the overall cache memory consumption?
- How does *LatCache* perform across different loads, cache sizes, and number of AAs?

- Do the benefits of the *LatCache* algorithm translate to production-scale systems?

6.1 Workloads and Methodology

Our simulations utilize six real traces collected from high-traffic allocator nodes across various zones. These traces encompass a diverse range of unique request types, with counts varying between 500 and 1.7k. Additionally, certain zones exhibit notably higher request rates and burstiness than others. Each trace spans a 24-hour horizon and includes all VM requests, providing information such as request id, arrival time, and request traits. The trace also contains information about the actual processing time of each request, which are used to simulate hit and miss times. We have also deployed Kamino with *LatCache* in all production zones and measure performance in a subset of representative zones.

6.2 Effectiveness of *LatCache* on Latency

We first use our simulator to understand the impact of *LatCache* on reducing average and tail VM allocation request latency. We compare it against the load-balancing algorithm used by the state-of-the-art VM allocator Protean. We also compare it against classical cache-oblivious load-balancing algorithms (random assignment and round-robin). Finally, we compare it against a cache-aware solution consisting of using consistent-hashing assignment augmented with work-stealing to reduce hotspots [47]. Thus, we compare *LatCache* against:

(1) **Protean with a shared queue**. This consists of a single FIFO per allocator node that receives all requests, and AAs pull requests from this queue when idle. This is the algorithm previously used in Protean.

(2) **Random assignment (*Random*)**. Assigns the incoming requests uniformly at random to one of the AAs.

(3) ***Round-Robin***. Assign the incoming requests to the AAs in a Round-Robin fashion.

(4) ***Hash+WS***. This uses **consistent hashing + work stealing**, employed in other systems, to determine data location and task execution [47]. The assignment to an AA is computed by hashing the request traits, ensuring that requests with the same cache objects are assigned to the same AA [21, 32], which promotes data locality and reduces cache replication. Additionally, when an AA is idle, it steals a pending request from the AA with the largest queue, if any, which helps reduce hot spots caused by skewed request distributions.

Unfortunately, the scheduling algorithms of other large-scale VM allocators [36, 41] have not been openly documented, preventing us from recreating and evaluating them.

We also evaluate two versions of *LatCache*: In addition to *LatCache-rule*, the algorithm as described in §4, we consider the variant *LatCache-request* that uses a simpler estimation of the processing time of a request based only on its presence at the top-level of the cache of the given AA, i.e., on a top-level miss, instead of looking up the presence of individual rule on the lower-level cache as in Algorithm 2, it assumes that none of them is present in the lower-level cache. Its main advantage is that it requires fewer and simpler (i.e., to a single level) cache lookups. For this analysis, we fix the number of AAs on each node to 4 to match our current production system. We also reserve the total cache memory based on production configuration (up to 58% of the node's total memory); in §6.6, we vary these parameters in our simulated environment.

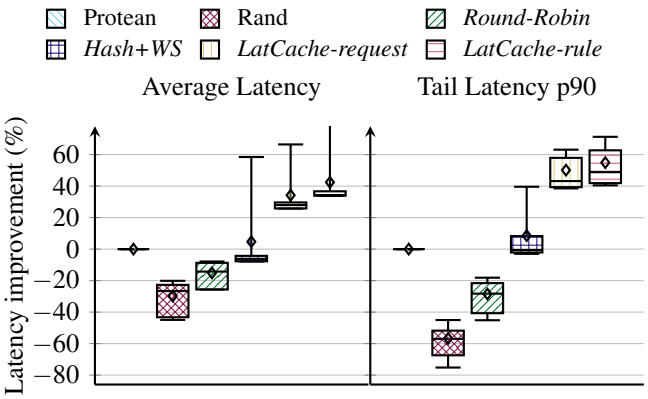


Figure 6: Latency improvements for each algorithm

Fig. 6 illustrates the percentage improvement in average and 90th percentile tail allocation latency of the different algorithms compared to the baseline Protean. Standard load-balancing algorithms *Random* and *Round-Robin* exhibit approximately 20% worse performance on average compared to the baseline. The cache-aware *Hash+WS* algorithm, which promotes request co-location plus simple load-balancing, achieves only 4.4% average latency improvement with high variance and 9.1% improvement in tail latency. In contrast, our *LatCache-request* and *LatCache-rule* algorithms achieve significant improvements in both average and tail latency compared to all other algorithms, with over 50% tail latency improvement over the baseline Protean. Notably, lighter-weight *LatCache-request* is competitive with *LatCache-rule*, despite ignoring lower-level cache content during processing time estimation.

While request allocation latency is our main metric, the superiority of our algorithms also holds in terms of throughput, albeit in a less striking manner. Nonetheless, in periods of bursts of requests, we consistently see a 2x throughput improvement compared to Protean. Fig. 7 plots the throughput profiles of the algorithms in a window with a burst of requests (for clarity's sake, *Random*, *Round-Robin*, and *LatCache-request* were omitted).

6.3 Explaining why *LatCache* works well

Table 3 (second column) shows the top-level cache hit rates for various algorithms. *LatCache-rule* and *LatCache-request* achieve the highest hit rates ($\approx 94\%$), significantly higher than the cache-friendly *Hash+WS* ($\approx 87\%$) and the cache-oblivious Protean, *Random*, and *Round-Robin* ($\approx 81\%$). This validates *LatCache*'s superior ability to promote data locality,

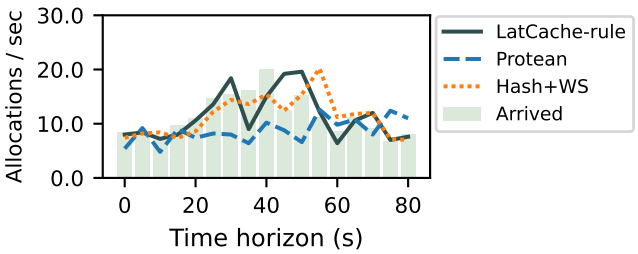


Figure 7: Throughput Comparison. Results are shown for two consecutive bursts of requests.

Algorithms	Hit ratio (%)	Norm. cache memory
Protean	80.7 ± 2.9	1.00 ± 0.00
Random	81.1 ± 3.0	0.98 ± 0.00
Round-Robin	80.6 ± 3.0	1.00 ± 0.00
Hash+WS	87.4 ± 3.1	0.94 ± 0.05
LatCache-request	93.1 ± 3.2	0.85 ± 0.09
LatCache-rule	95.0 ± 2.3	0.77 ± 0.06

Table 3: Cache hit ratio and normalized memory use.

as a consequence of careful latency predictions that use both current and projected cache states.

Next, Fig. 8 shows the breakdown of the average request latency of the algorithms into the time waiting in the queue of an AA and request processing time. *LatCache* algorithms have the smallest processing times, due to their high cache hit rates. However, the major part of *LatCache*'s latency improvement comes from a significant reduction in waiting time. Two main reasons contribute to lower wait times: 1) higher hit rates have a compounding effect on the multiple requests in a queue, leading to faster turnaround; 2) the latency-driven design with careful latency prediction accomplishes an effective request load-balancing across the AAs, as supported by Theorem 1. This *highlights the benefits of combining latency-driven scheduling with cache-awareness*.

6.4 Estimation accuracy and its impact

As mentioned earlier, the two main sources of inaccuracy that might affect the total latency estimation quality are: (1) (top-level) hit and miss event prediction; (2) estimation of the individual hit and miss times of the requests. We now quantify the accuracy of these estimations and their impact on the quality of the assignments of *LatCache-rule*.

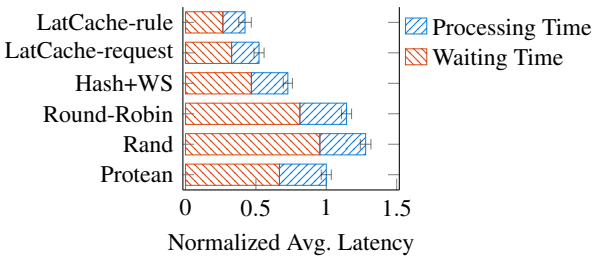


Figure 8: Latency breakdown.

First, our optimistic prediction for hit/miss events using the augmented cache state is highly accurate (99.1% across both cache levels). Given that hit times are significantly lower than miss times (average miss/hit time ratio: 5–10) and that hit/miss predictions directly affect queue latencies estimates, this level of accuracy is crucial for effective assignments.

Regarding the estimation of hit and miss times, we have an average prediction error of 29%. While the error is not small, using these estimates in conjunction with the accurate hit/miss event predictions leads to good allocation decisions: *LatCache-rule* selects the best allocator for 91.9% of the requests (best allocator = lowest actual total latency for the incoming request). Even when the best allocator is not selected, the scheduling decision is nearly optimal – the total latency percentage-gap between the selected allocator to the best allocator is only 2.3% on average. To put these numbers into perspective, a naive version of *LatCache-rule* that ignores the rule-level cache, future hit prediction, and allocators' remaining processing time chooses the best allocator only 65.4% of the time and the resulting total latency can be even 2x more than that of the best allocator.

We note that it is certainly possible to use more sophisticated machinery for estimating hit and miss times, such as applying machine learning techniques [12]. We observe, however, that the added value is not substantial. To quantify the possible gains, we ran *LatCache-rule* fed with perfect predictions of hit and miss times, and observed an additional improvement of 1.1% in tail allocation latency (vs. $\approx 50\%$ improvement that both versions of our algorithm have compared to the baseline). We conclude that the absolute hit and miss time predictions matter less than predicting the actual hit/miss event as well as capturing key elements of the allocation process, particularly the hierarchical-cache dynamics (see §4.2.2-4.2.3). The net outcome is that Kamino's scheduler has a simple yet effective and robust latency estimation that makes near-optimal allocation decisions.

6.5 Reduction in Memory Footprint

Table 3 presents the average cache memory size used by the algorithms, normalized to the baseline Protean. All cache-oblivious algorithms (Protean, *Random*, and *Round-Robin*) exhibit very similar memory occupation. In contrast, the cache-friendly *Hash+WS* demonstrates reduced usage (94% of Protean's memory usage), while our cache-aware *LatCache-request* and *LatCache-rule* show significantly lower usage (85% and 77% of Protean's, respectively). This reduction is attributed to the *LatCache* algorithms, which enhance cache hit rates and facilitate better reuse of cached rule objects by co-locating similar requests on the AAs. Notably, *LatCache-rule* achieves the smallest memory usage by promoting such co-location at the rule level, leveraging information about the objects present on the lower cache level. Overall, *LatCache*'s memory overhead is negligible: cache states are probed at runtime, and hit/miss latencies are already

tracked for monitoring. Reduced memory use effectively allows to increase the number of AAs on the same node by decreasing the required cache-per-AA to store the necessary items, a particularly important consideration in our production system (see §2.2.3).

6.6 Sensitivity to System Parameters

Next, we analyze the performance of the algorithms across different system parameters. For brevity, we drop *Round-Robin* and *Random* from the analysis due to their highly suboptimal performance observed earlier.

Sensitivity to the number of AAs: We vary the number of AAs (R) on a node while keeping the total memory cache (M) fixed, thereby changing the per-AA cache memory (M/R). As shown in Fig. 9a, Protean and *Hash+WS* algorithms scale poorly with increasing AA count. The benefits of parallelism from additional AAs are negated by cache fragmentation, as smaller private caches reduce effectiveness. In contrast, *LatCache-request* and *LatCache-rule* algorithms sustain their performance, with reduced per-agent cache having a smaller impact, consistent with their lower memory usage.

Sensitivity to allocation request load: We evaluate the performance of the algorithms under high load. We reduce the requests' interarrival time by multiplying each request's arrival time by a parameter $\varepsilon \leq 1$. The lower this ε parameter, the higher the frequency of requests. For analysis, we pick ε values that increase request frequency by 25%, 50%, 75%, and 100%. Fig. 9b shows tail latency improvement over Protean baseline. As shown, the benefits of cache-aware algorithms over Protean increase as the load increases, with *LatCache* algorithms exhibiting scalability and consistent gains over *Hash+WS* across all loads.

Sensitivity to cache size: We evaluate the performance of the algorithms under varying system memory for caching across AAs. Fig. 9c shows tail latency gains over Protean for each cache size, where 100% represents the size used in previous experiments. First, even with smaller cache sizes, cache-friendly algorithms like *Hash+WS*, *LatCache-request*, and *LatCache-rule* outperform the cache-oblivious Protean. Furthermore, *LatCache* algorithms consistently exceed both Protean and *Hash+WS*, even under limited cache memory. Second, for larger reserved cache sizes, *LatCache-request* and *LatCache-rule* still outperform Protean and *Hash+WS*, albeit with more modest gains. The gains diminish as larger caches accommodate more objects, reducing the importance of data locality.

6.7 Performance on Production Zones

We evaluate Kamino's performance impact with *LatCache* in production on a subset of zones. We deploy the *LatCache-request* algorithm due to its simpler integration with the current cache API and plan to roll out *LatCache-rule* later. We collect performance metrics across 5 production zones, comprising tens of thousands of nodes, over 15 days before and after deployment with *LatCache-request* (Kamino-*LatCache*).

Latency	Protean	Kamino-*LatCache*	Improve.
Avg. (ms)	185.6 ± 20.4	146.3 ± 17.4	21.1%
90p (ms)	378.8 ± 90.8	333.5 ± 64.7	11.9%

Table 4: Allocation latencies before (Protean) and after deploying Kamino.

Impact on latency. We compare allocation latencies before (Protean) and after deploying Kamino-*LatCache*. Table 4 shows average and 90th percentile latencies with standard deviations. We observe a 21.1% reduction in average latency and an 11.9% reduction at the 90th percentile, aligning with simulation results. However, modest gains arise from factors not modeled in the simulator, such as conflicts and retries when multiple AAs place VMs on the same physical machine, causing infeasible placements. Fig. 10 shows disaggregated latency performances of Protean and Kamino, highlighting consistent gains across all zones.

Impact on cache hit rate. To display the impact of Kamino-*LatCache* on cache hit ratios, we aggregate the measurements of the five zones and plot in Fig. 11 the hit rate over 2 months; the dotted line (in early Feb.) marks where Kamino-*LatCache* replaces Protean. Overall, cache hits increase from 80% to 86.6% on average across the five zones; thus, the total cache misses reduced by 33%. Two main factors contribute to the differences in cache hit gains between the simulation and production systems: (a) varying memory cache sizes across the fleet and (b) dynamic workload changes over 15 days on production compared to 24-hour simulation traces.

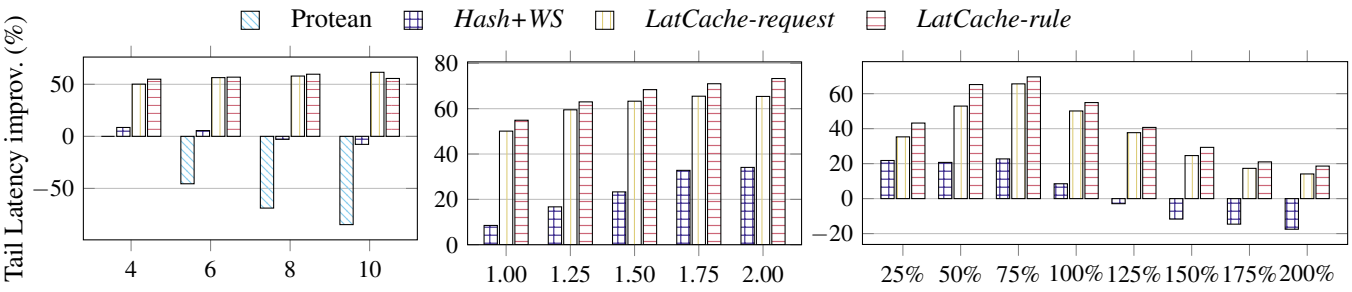
We also observed that a significant reduction in cache misses reduces the number of AA nodes, lowers contention and communication overhead, and minimizes allocation retries, yielding more efficient request handling and improved system stability.

Reduction in CPU and memory resource usage. We evaluate Kamino-*LatCache*'s impact on resource efficiency in allocator nodes, focusing on aggregate working memory and CPU usage across AAs per node. On average, Kamino-*LatCache* reduces memory usage by 17% and CPU usage by 18.6%, consistent with simulation results (Table 3). By promoting data locality through co-locating similar requests on AAs, Kamino-*LatCache* reduces cache content replication, saving memory. This reduction also lowers CPU usage, as less computation is needed for cache updates. Measurements confirm this, showing a 10–20% drop in cache updates per minute.

7 Related Work

In addition to our earlier overview of allocator systems, we survey related work in resource management.

Request scheduling and cache-awareness. Prior research has focused on efficient request scheduling and load-balancing to minimize latency, especially for sub-second latencies. They include using OS techniques like per-CPU caches, optimizing CPU allocation for application threads,



(a) Number of AAs (b) Trace scale up factor (c) Cache Sizes

Figure 9: Sensitivity to System Parameters: Tail Latency p90 improvement (%)

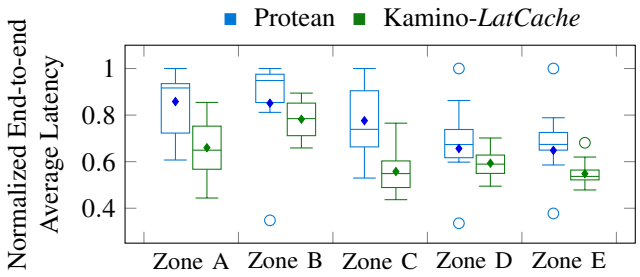


Figure 10: Production: Average latencies before and after deployment, normalized by the largest latency in the zone.

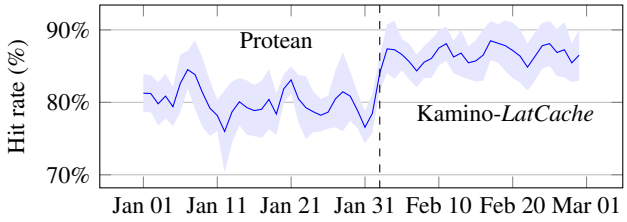


Figure 11: Production: Hit rates before and after deployment day. Dashed line indicates deployment day.

reducing queue wait times and scheduling costs [20, 33, 34]. Web services research explores static and adaptive scheduling strategies to address latency and throughput challenges [23, 28, 39]. However, techniques like preemptive work suspension and network queue polling are unsuitable for VM request scheduling. Cache-aware scheduling has been explored, with Google's search system using hypergraph partitioning for load-balancing [4], and [37] proposing similar methods for Facebook's web traffic and storage sharding. Other studies improve latencies through dynamic cache partitioning of shared caches or adapting cache policies, often using hit rates as latency proxies [10, 48]. To our knowledge, Kamino is the first publicly reported VM allocation framework integrating cache awareness and latency considerations for scheduling.

Cluster and resource scheduling. A huge body of work addresses sharing distributed resources across computations. Frameworks like Omega [36], Twine [41], and others [13, 14, 17, 18, 31, 44, 46] optimize resource utilization for jobs or tasks. Similar approaches target container scheduling (e.g., Kubernetes [27, 29]) and VM deployment (e.g., Protean [16]). Others focus on resource allocation with service-level objectives (SLOs) [19] and dynamic allocation policies leveraging predictions and continual optimization [6, 11, 12, 22, 24, 30, 35, 38, 40, 50]. However, no prior work integrates caching with resource efficiency for VM or container allocation.

Algorithms for scheduling and paging. There is a rich body of theoretical work on page replacement algorithms for single-cache systems [3, 8, 26] and scheduling algorithms for latency minimization, though most focus on systems without caches (e.g., job scheduling models) [5, 7]. The intersection of these areas remains underexplored. [45] addresses scheduling in multi-cache systems but focuses on minimizing cache misses.

8 Conclusion

We design and implement *Kamino* – a latency-driven and cache-aware VM request scheduling framework. Our algorithm, *LatCache*, relies on clever estimates of latency based on the cache state. An extensive simulation study with real traces, followed by measurements from large-scale production zones show increased resource efficiency and sizeable reductions in end-to-end latency. One interesting direction for future work is examining incorporating fast and persistent storage devices such as SSDs to serve as a secondary cache to alleviate memory bottlenecks [4, 42, 49].

Acknowledgments

We would like to thank our shepherd, Akshitha Sriraman, and the anonymous OSDI reviewers for their detailed and insightful feedback that has improved the presentation of the paper. We also want to thank Esaias E Greeff, Ricardo Binachini and Johannes Gehrke for helpful discussions.

Sudarsun Kannan and David Domingo were supported in part by NSF CNS grants 1910593, 2231724, 2319944, and NSF CAREER grant 2443438. A part of this work also used the experimental platform funded by the NSF II-EN grant 1730043.

A Appendix

A.1 Applying *LatCache* Principles Beyond VM Allocators

To evaluate the generalizability and feasibility of applying *LatCache* 's principles—combining locality awareness with latency-aware queuing—to systems with hierarchical data tiers, we built a proof-of-concept prototype that applies *LatCache* scheduling to log-structured merge (LSM) trees.

To test latency- and cache-aware scheduling beyond virtual machine allocation, we adapt LatCache's principles to optimize lookups in Log-Structured Merge (LSM) Trees [1, 2]. Our prototype shows reduced lookup latency and lower computational overhead. Below, we briefly review LSM trees, followed by our approach for latency and cache-sensitive scheduling.

LSM background. LSM trees underpin many key–value stores, including LevelDB and RocksDB [1, 2]. An LSM tree is organized into multiple levels that grow in size from top (level 0) to bottom. New writes buffer in memory and flush as sorted table files at level 0. As levels fill, files merge and compact downward. This design yields high write throughput but can slow reads: each lookup may scan levels sequentially until the key is found or all levels are exhausted. To mitigate read amplification, each SSTable maintains a Bloom filter—a compact, probabilistic structure that can definitively rule out a key or indicate it might be present. By checking the Bloom filter before probing an SSTable, lookups skip files unlikely to contain the key, reducing unnecessary I/O and cache pollution. However, Bloom filters can produce false positives and still require sequential filter checks across levels, so lookups may probe multiple levels and incur latency and cache churn.

Latency and cache-sensitive request scheduling. Our design uses Bloom-filter–based locality hints in LSM trees to direct lookups to the levels most likely to contain the target key and incorporates per-level latency estimates to decide when parallel searches are warranted. This approach reduces redundant computation and cache pollution while still exploiting parallelism.

In our LevelDB-based LSM prototype, this approach achieves a 22% reduction in average lookup latency compared to searching serially across LSM tiers. Although this prototype uses a single worker thread for each level, increasing CPU assignments and refining latency models should yield further gains. These preliminary results confirm that combining locality awareness with latency-aware queuing strikes an effective balance between throughput and resource efficiency in hierarchical storage. Our future work will characterize the trade-offs under mixed read–write and tail-latency–sensitive workloads and generalize *LatCache* 's benefits across diverse architectures.

References

[1] Google/leveldb: Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values.

[2] Rocksdb: A persistant key-value store.

[3] Susanne Albers. Online algorithms: a survey. *Math. Program.*, 97(1-2):3–26, 2003.

[4] Aaron Archer, Kevin Aydin, MohammadHossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. Cache-aware load balancing of data center applications. 2019.

[5] Nikhil Bansal. *Minimum Flow Time*, pages 531–533. Springer US, Boston, MA, 2008.

[6] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, et al. Virtual machine allocation with lifetime predictions. *Proceedings of Machine Learning and Systems*, 5:232–253, 2023.

[7] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs. *Flow Time Minimization*, pages 320–322. Springer US, Boston, MA, 2008.

[8] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[9] Ben Berg, Daniel Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, 2020.

[10] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching–dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, 2018.

[11] Niv Buchbinder, Yaron Fairstein, Konstantina Mellou, Ishai Menache, and Joseph Naor. Online virtual machine allocation with lifetime and load predictions. *ACM SIGMETRICS Performance Evaluation Review*, 49(1):9–10, 2021.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[13] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110, 2015.

[14] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, 2016.

[15] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[16] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for Fine-Grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[18] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[19] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, 2016.

[20] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[22] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. Selftune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1097–1114, 2023.

[23] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

[24] Jianheng Ling, Pratik Worah, Yawen Wang, Yunchuan Kong, Chunlei Wang, Clifford Stein, Diwakar Gupta, Jason Behmer, Logan A. Bush, Prakash Ramanan, Rajesh Kumar, Thomas Chestna, Yajing Liu, Ying Liu, Ye Zhao, Kathryn S. McKinley, Meeyoung Park, and Martin Maas. Lava: Lifetime-aware vm allocation with learned distributions and adaptation to mispredictions. *Proceedings of Machine Learning and Systems*, 2025.

[25] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[26] Alejandro López-Ortiz and Alejandro Salinger. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 113–127, New York, NY, USA, 2012. Association for Computing Machinery.

[27] Marko Luksa. *Kubernetes in action*. Simon and Schuster, 2017.

[28] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, 2022.

[29] Tarek Menouer. Kcss: Kubernetes container scheduling strategy. *The Journal of Supercomputing*, 77(5):4267–4293, 2021.

[30] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. Ras: Continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS*

28th Symposium on Operating Systems Principles, pages 505–520, 2021.

[31] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 69–84, 2013.

[32] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[33] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[34] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[35] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, and Ishai Menache. Kerveros: Efficient and scalable cloud admission control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 227–245, Boston, MA, July 2023. USENIX Association.

[36] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[37] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. Social hash: An assignment framework for optimizing distributed systems operations on social networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 455–468, Santa Clara, CA, March 2016. USENIX Association.

[38] Junjie Sheng, Yiqiu Hu, Wenli Zhou, Lei Zhu, Bo Jin, Jun Wang, and Xiangfeng Wang. Learning to schedule multi-numa virtual machines via reinforcement learning. *Pattern Recognition*, 121:108254, 2022.

[39] Bhavana Vannarth Shobhana, Srinivas Narayana, and Badri Nath. Load balancers need in-band feedback control. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 76–84, 2022.

[40] Suruchi Talwani, Jimmy Singla, Gauri Mathur, Navneet Malik, NZ Jhanjhi, Mehedi Masud, and Sultan Aljahdali. Machine-learning-based approach for virtual machine allocation and migration. *Electronics*, 11(19):3249, 2022.

[41] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803, 2020.

[42] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[43] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020.

[44] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[45] Rahul Vaze and Sharayu Moharir. Paging with multiple caches. In *2016 14th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, pages 1–8, 2016.

[46] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[47] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 449–462, USA, 2020. USENIX Association.

[48] Gang Yan and Jian Li. Towards latency awareness for content delivery network caching. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 789–804, 2022.

[49] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *2022*

USENIX Annual Technical Conference (USENIX ATC 22), pages 1021–1036, 2022.

[50] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. History-Based harvesting of spare cycles and storage in Large-Scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 755–770, 2016.

ZEN: Empowering Distributed Training with Sparsity-driven Data Synchronization

Zhuang Wang*, Zhaozhuo Xu◇*, Jingyi Xi†*, Yuke Wang, Anshumali Shrivastava, and T. S. Eugene Ng
Rice University, ◇Stevens Institute of Technology, †Unaffiliated

Abstract

Distributed training is the de facto standard to scale up the training of deep learning models with multiple GPUs. Its performance bottleneck lies in communications for gradient synchronization. Although high tensor sparsity is widely observed, the optimal communication scheme to fully leverage sparsity is still missing. This paper aims to bridge this gap. We first analyze the characteristics of sparse tensors in popular models to understand the fundamentals of sparsity. We then systematically explore the design space of communication schemes for sparse tensors and find the optimal ones. These findings give a new understanding and inspire us to develop a holistic gradient synchronization system for sparse tensors called ZEN. We demonstrate that ZEN can achieve up to 5.09× speedup in communication time and up to 2.48× speedup in training throughput compared to the state-of-the-art methods.

1 Introduction

In recent years, deep learning (DL) models have achieved remarkable empirical performance in real-world applications, such as language processing [6, 22, 32, 40, 64, 67] and recommendation systems [26, 42]. With the ever-growing size of models and training datasets, distributed training has become the norm for model training with multiple GPUs [35, 47, 49, 59, 71]. The synchronization of gradient tensors from different GPUs is commonly required in distributed training. In data parallelism [29, 35, 59], the training dataset is partitioned across multiple GPUs, so the corresponding gradient tensors must be synchronized to maintain model consistency. In tensor parallelism [62], individual layers of a DL model are sharded over multiple GPUs, which synchronize the gradient tensors during backward propagation for the gradient computation of subsequent layers. Furthermore, it is common practice to train large models [17, 49, 58] with a mix of data parallelism and other parallelism strategies, such as pipeline parallelism [28, 47], tensor parallelism [62], and ZeRO [55, 81]. These training workloads must also synchronize the gradient tensors across GPUs.

The major efficiency bottleneck of distributed training lies in the communication for gradient synchronization [23, 72, 73, 80]. Recent hardware developments have greatly improved the computation efficiency of model training. These advancements increase the frequency of gradient synchronization in distributed training and shift more burdens to network systems. However, network upgrades have not kept up with the improvements in computation [43, 50, 51, 80], exacerbating the tension between computation and communication.

Several communication schemes have been developed to alleviate communication bottlenecks in distributed training by fully utilizing the network bandwidth. For example, Ring-Allreduce [53] is provably bandwidth-optimal in homogeneous GPU clusters and it is widely used in collective communication libraries such as MPI [18], NCCL [4], and MSCCL [5]. BytePS [29] is a communication-optimal architecture for heterogeneous GPU/CPU clusters. However, both Ring-Allreduce and BytePS assume that tensors to be synchronized are dense, ignoring their sparsity.

Recent work has shown that DL models exhibit a high degree of tensor sparsity during gradient synchronization. The sparsity comes from either natural gradient computation [16, 37, 42] or gradient compression sparsification algorithms [7, 38, 73]. On the one hand, because model training can focus on updating a subset of parameters instead of all of them [24, 78], some of the gradient tensors in DL models are naturally sparse and most of the gradients in these tensors are zero. For instance, the embedding table in the widely deployed Deep Learning Recommendation Models [70] can reach more than 93% sparsity; Graph adjacency matrix in Graph Neural Networks [69] usually retains the sparsity more than 99%; Word embeddings of those popular natural language processing models, such as NMT [40] and LSTM [44], would achieve more than 97% sparsity. On the other hand, to address the communication bottleneck, a plethora of sparsification algorithms [7, 38, 39, 60, 61, 63, 73] are proposed

*These authors contributed equally.

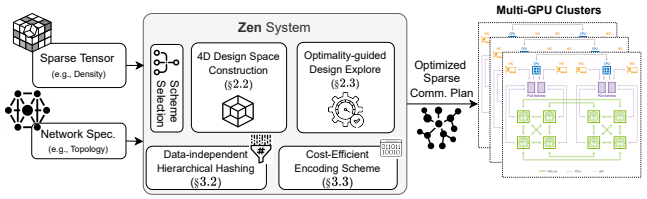


Figure 1: ZEN System Overview.

to reduce synchronization traffic by selecting a subset of the original stochastic gradients. They can save up to 99% of the gradient exchange while maintaining model accuracy [12, 39].

Leveraging the notable sparsity can significantly reduce traffic volume for gradient synchronization and shorten the communication time in distributed training. We can represent non-zero gradients with a sparse format and denote the tensor with a sparse format as a *sparse tensor*. Previous works, such as AGsparse [35], SparCML [56], and OmniReduce [23], have acknowledged this potential. They use various sparse formats and synchronization schemes. However, these approaches do not fully consider the fundamental characteristics of sparsity in DL models and lack understanding of the optimal scheme, resulting in suboptimal communication performance for gradient synchronization. To this end, this paper addresses the research problem: *What is the optimal communication scheme for gradient synchronization of sparse tensors in distributed training?*

To address this question, it is essential to revisit the fundamentals of the tensor sparsity in DL models. We first unveil the characteristics of sparse tensors in popular models [22, 26, 30, 40, 65, 79] across GPUs, iterations, and diverse training workloads. We then explore locations and distributions of non-zero gradients in tensors and the changes observed in sparse tensors before and after aggregation under varying configurations.

In light of these characteristics of sparse tensors, we systematically explore the design space of synchronization schemes for sparse tensors. Four elemental dimensions are proposed, including communication, aggregation, partition, and balance, to construct diverse synchronization schemes, revealing that existing schemes [23, 35, 56] can be described within this framework. This leads to a proof of the existence of communication-optimal schemes to synchronize sparse tensors in distributed training.

Based on these findings, we develop a sparse gradient synchronization system, called ZEN (Figure 1), which achieves a near-optimal communication time. Specifically, ZEN takes the tensor sparsity and network specification as input and pinpoints the optimized sparse communication plan as output. Our key design insight lies in framing the challenge of achieving optimal schemes as a mathematical problem. We propose a data-independent solution to address this challenge, eliminating costly data-dependency overhead. This solution achieves high efficiency by introducing a novel hierarchical hashing algorithm that attains superior performance on GPUs while ensuring provably near-optimal results without any information loss. ZEN also incorporates an efficient encoding scheme to minimize index representation overhead, irrespective of the random distribution of non-zero gradients after applying the hierarchical hashing algorithm. We evaluate ZEN across various DL training workloads, where tensor sparsity arises from natural computation or gradient compression. ZEN achieves up to 5.09× speedup in communication time and up to 2.48× speedup in training throughput over existing methods for sparse tensor synchronization.

Table 1: Three DL models with natural tensor sparsity.

Model	MLP Size	Embedding Size	Embedding Density
LSTM [44]	20M	406M	1.13%
DeepFM [26]	68M	214M	2.80%
NMT [40]	31M	112M	2.47%

Table 2: Configurations of three LLMs. AH: attention heads.

Model	Hidden size	Intermediate	#Layers	#AH
Llama3.2-3B [22]	3,072	8,192	28	24
OPT2.7B [79]	2,560	10,240	32	32
Gemma2-2B [65]	2,304	9,216	26	8

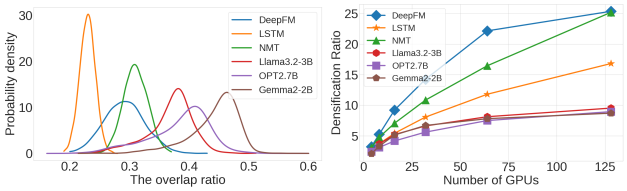
The key contributions of this paper are as follows.

- We analyze the characteristics of sparse tensors in popular DL models to reveal and understand the fundamentals of gradient sparsity in distributed training.
- We systematically explore the design space of sparse communication schemes and pinpoint the optimal ones.
- We develop ZEN, a gradient synchronization system for sparse tensors that achieves near-optimal communication time using a data-independent hierarchical hashing algorithm and an efficient encoding scheme.
- Evaluation reveals that ZEN achieves evident throughput improvement compared to the state-of-the-art methods across diverse training workloads.

2 Analysis of Sparse Tensor Synchronization

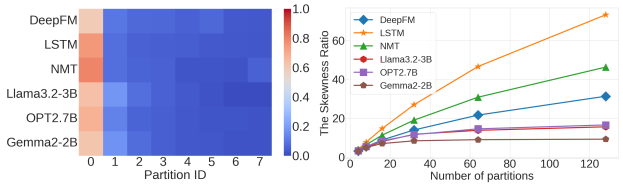
2.1 Characteristics of Sparse Tensors

This section analyzes the characteristics of sparse tensors from both natural gradient computation and sparsification algorithms. For natural tensor sparsity, we will study the gradient tensors from the embedding layers of the three DL models listed in Table 1. For the tensor sparsity from gradient compression, we will study three large language models (LLMs) listed in Table 2 and apply DGC [39], one of the most popular sparsification algorithms, to select the top 5% gradients from their gradient tensors [8, 27]. Detailed workloads for the models can be found in §5.1.



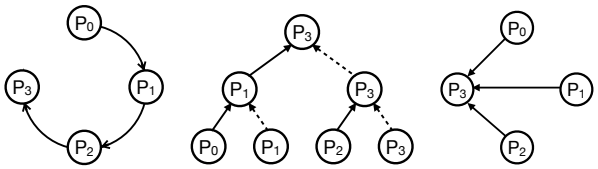
(a) (b)

Figure 2: The characteristics of sparse tensors in DL models. (a) shows that the overlap ratio of sparse tensors varies; (b) shows that tensors have higher density after aggregation.



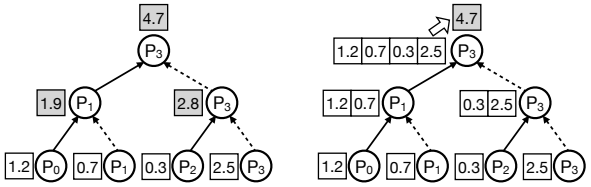
(a) (b)

Figure 3: The distribution of non-zero gradients is skewed. (a) The heatmap of non-zero gradients distribution; (b) The skewness ratio.



(a) Ring (b) Hierarchy (c) Point-to-point

Figure 4: An illustration of three communication patterns with four GPUs. GPU P_3 aggregates the data from all GPUs.



(a) Incremental aggregation. (b) One-shot aggregation.

Figure 5: An Illustration of two aggregation patterns with Hierarchy. The gradients on each GPU are from the same parameter and 4.7 is the final aggregated result.

Definition 1 (Dense tensor)**.** *We define the original gradient tensor in a DL layer as a dense tensor.*

We define *density* of a gradient tensor as the percentage of its non-zero gradient values. We can represent a gradient tensor in a sparse format when many parameters have zero gradients. A typical sparse format is coordinate lists (COO) that store a list of non-zero gradients and a list of the corresponding indices [23, 77].

Definition 2 (Sparse tensor)**.** *We define a gradient tensor in a sparse format as a sparse tensor.*

We denote the size of a dense tensor G by M, its density is d_G, and the training involves n nodes. For simplicity, we assume that each node has only one GPU in this section.

C1: The overlap of sparse tensors varies. Similar to dense tensors, sparse tensors are aggregated during synchronization. When aggregating dense tensors, the indices of gradients from different GPUs are identical. However, due to different batches as input for training on different GPUs, the indices of non-zero gradients in a sparse tensor are unknown *a priori*. They can have overlaps, but how much they overlap depends on many factors, such as the DL model, the training dataset, and the batches. We define the overlap ratio [68] to quantify this overlap between two sparse tensors.

Definition 3 (The overlap ratio)**.** *Given two sparse tensors and their sets of indices for non-zero gradients are I_1 and I_2, respectively, their overlap ratio is defined as $\frac{|I_1 \cap I_2|}{\min\{|I_1|,|I_2|\}}$, where $|\cdot|$ is the cardinality of a set.*

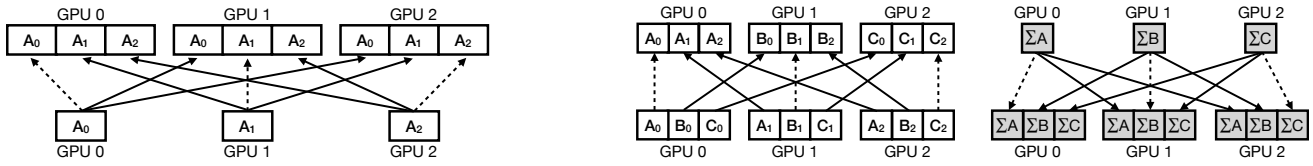
Figure 2a shows the probability density function (PDF) of the overlap ratios for the six DL models studied in the paper. The overlap ratio in a model is approximately normally distributed, and it is in a wide range. In addition, different models have different distributions of overlap ratios.

C2: The tensor size after aggregation varies. When aggregating dense tensors, the tensor sizes before and after aggregation remain the same. However, when aggregating sparse tensors, the unknown overlaps of sparse tensors lead to varying tensor sizes after aggregation. Because the aggregation involves sparse tensors from multiple GPUs, we denote d_G^n as the density after the aggregation of tensors from n GPUs. We observe that sparse tensors get denser after aggregation. We define the densification ratio to quantify this characteristic.

Definition 4 (The densification ratio)**.** *Given a dense tensor G, its densification ratio is defined as $\gamma_G^n = \frac{d_G^n}{d_G}$.*

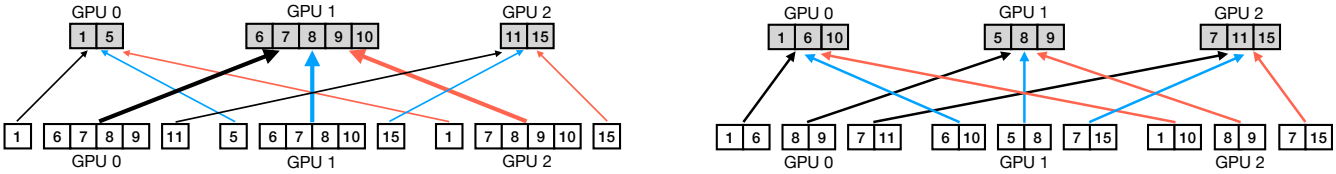
Figure 2b presents the average densification ratio γ_G^n to the number of GPUs for the DL models studied in this paper. The densification ratio increases with the number of GPUs, demonstrating that tensors have a higher density after aggregation. We can also see that the densification ratio is smaller than the number of GPUs, i.e., $\gamma_G^n < n$. It suggests that the indices of non-zero gradients in sparse tensors from different GPUs are partially overlapped.

C3: The distribution of non-zero gradients is skewed. When evenly splitting a dense tensor into multiple partitions, we observe that most of the non-zero gradients are in one of them. For example, with eight partitions, more than 60% of the non-zero gradients are in the first partition in the six DL



(a) Centralization (b) Parallelism

Figure 6: An illustration of the two partition patterns with Point-to-point. In (a), each tensor is communicated as a whole and each GPU receives all the tensors. In (b), each tensor is split into three partitions; the same partition from different GPUs is sent to the same place, and the aggregated results are then sent back to all GPUs.



(a) Imbalanced communication (b) Balanced communication

Figure 7: An illustration of balance patterns with Point-to-point and Parallelism. Each GPU has six non-zero gradients and the numbers are their indices. In (a), four gradients from each GPU are sent to GPU 1. In (b), each GPU sends two gradients to other GPUs, and communications are well-balanced. However, it is non-trivial to achieve such balanced communications.

models. Figure 3a shows the percentage heatmap of the non-zero gradients in each partition. We define the skewness ratio to quantify the skewed distribution of non-zero gradients.

Definition 5 (The skewness ratio)**.** *Given a dense tensor G and we evenly divide G into disjoint n partitions, denoted as $\{G_1, \cdots, G_n\}$, the skew ratio of G with n partitions is defined as* $\frac{\max_{i\in[n]}\{d_{G_i}\}}{d_G}$.

Figure 3b presents the skewness ratios of gradient tensors in the DL models studied in this paper. They are significant in all six models. For example, when we evenly split the gradient tensor from the embedding table in LSTM into 128 partitions, the skewness ratio is over 70. It indicates that more than half of the non-zero gradients are in the same partition.

2.2 Elemental Dimensions for Synchronization

Synchronization for dense tensors has been extensively studied [29, 34, 59, 66]. This section will explore the design space to construct synchronization schemes for sparse tensors. Given a tensor, the outcome of its synchronization is that gradients with the same indices are aggregated and all GPUs have identical aggregated results. We will discuss four dimensions that construct a synchronization scheme for sparse tensors.

Communication dimension. There are typically three communication patterns for synchronization: 1) `Ring`, 2) `Hierarchy`, and 3) `Point-to-point`. They are illustrated in Figure 4 with an example in which there are four GPUs and GPU P_3 aggregates the data from all GPUs. In Ring, all GPUs form a ring structure. P_0 first sends its data to P_1, which then passes the data along with its own data to P_2 and so on until P_3 receives all the data. In Hierarchy, all GPUs form a hierarchical structure and P_3 is the root. There are two stages in Figure 4b. In the first stage, P_0 sends its data to P_1 and P_2 sends its data to P_3. In the second stage, P_1 sends the data from both its own and P_0 to P_3. In Point-to-point communication, the other three GPUs directly send data to P_3.

Aggregation dimension. A communication pattern can have multiple communication stages and there are two options for aggregation: 1) `Incremental aggregation`, i..e, aggregate tensors at each stage; and 2) `One-shot aggregation`, i.e., only aggregate tensors from all GPUs after the last stage. These two options lead to different amounts of traffic volume in different communication stages because of C1 and C2 discussed in Section 2.1. In the example illustrated in Figure 4, Ring has three stages and Hierarchy has two stages. Figure 5 displays an example with Hierarchy as the communication pattern. When P_1 receives a tensor from P_0, it has two tensors. P_1 can either aggregate the two tensors and send the aggregated result to P_3, as shown in Figure 5a; or it can just send the concatenated tensor to P_3, as shown in Figure 5b.

Partition dimension. There are two partition patterns to ensure that all GPUs have the same aggregated results after synchronization: 1) `Centralization`, in which each tensor is communicated and aggregated as a whole; and 2) `Parallelism`, in which each tensor is decomposed into multiple partitions and each partition is communicated and aggregated separately. Figure 6 compares the two partition patterns with Point-to-point as the communication pattern. With Centralization, as shown in Figure 6a, each GPU sends its tensor as a whole to other GPUs. With Parallelism, as shown in Figure 6b, each GPU first decomposes its tensor into three

partitions and it requires two steps for synchronization. The first step aggregates the same partition from different GPUs in different places and the second step ensures that all GPUs have the aggregated results of all partitions.

Balance dimension. With Parallelism, the number of non-zero gradients in each partition can vary. Therefore, there are two patterns in terms of the traffic volume received at each GPU: 1) `Balanced communication`, in which the GPUs receive the same amount of data; and 2) `Imbalanced communication`, in which the traffic volumes received at different GPUs are greatly different. Figure 7 compares the two balance patterns among three GPUs with Point-to-point. There are 15 gradients in the tensor and six of them are non-zero. As shown in Figure 7a, four non-zero gradients are in the middle partition and they are sent to GPU 1. The traffic volume received at GPU 1 is 4× that received at GPU 0 and GPU 2. In Figure 7b, each GPU sends two non-zero gradients to other GPUs and the volume among them is well balanced. Naively partitioning a tensor can cause imbalanced communications due to C3 discussed in §2.1.

The four dimensions describe the design space of synchronization schemes for sparse tensors. Table 3 classifies existing schemes [23, 35, 56] based on their dimensions.

2.3 Optimal Synchronization Schemes

Next, we analyze optimal synchronization schemes for sparse tensors based on the four design dimensions. For convenience, we introduce two special schemes: ***Balanced Parallelism*** and ***Hierarchical Centralization***.

Definition 6 (Balanced Parallelism)**.** *It refers to the synchronization scheme characterized by [Point-to-point, Incremental aggregation, Parallelism, and Balanced communication].*

Definition 7 (Hierarchical Centralization)**.** *It refers to the synchronization scheme characterized by [Hierarchy, Incremental aggregation, and Centralization].*

Theorem 1 (Optimal schemes)**.** *To minimize communication time for sparse tensors, the optimal synchronization scheme is either Balanced Parallelism or Hierarchical Centralization.*

Proof. Theorem 1 is proven using Lemma 1 and Lemma 2. Here, we present the main intuitions and the detailed proof is in Appendix B.1.

Lemma 1. *When the partition pattern is fixed to Parallelism, the optimal scheme is Balanced Parallelism.*

There are three intuitions for Lemma 1: 1) Balanced communication outperforms imbalanced communication; 2) Point-to-point communication outperforms both Ring and Hierarchy by minimizing the traffic volume of unique gradients in the Parallelism partition pattern. and 3) Incremental aggregation outperforms One-shot aggregation as it reduces the traffic volume by aggregating the overlaps of sparse tensors.

Lemma 2. *When the partition pattern is fixed to Centralization, the optimal scheme is Hierarchical Centralization.*

There are two intuitions for Lemma 2: 1) when the partition pattern is fixed to Centralization, the search space is reduced to six candidates because the Balance dimension is not applicable. 2) Among these candidates, Hierarchical Centralization minimizes the traffic volume for overlapped gradients. We use an extreme case to demonstrate the second intuition. Suppose a non-zero gradient with index *idx* appears in all sparse tensors. In Point-to-point or One-shot aggregation, each GPU has to receive this gradient $n-1$ times. In Ring, the gradient from each GPU is aggregated at every stage and forwarded to the next GPU, causing each GPU to receive the gradient $n-1$ times. However, with Hierarchy and Incremental aggregation, each GPU only receives the gradient $\log n$ times.

Lemma 1 and Lemma 2 imply Theorem 1. □

Communication time analysis. We will analyze the theoretical communication time of the two schemes mentioned in Theorem 1. The symbols used in the analysis are listed in Table 4. We assume that each node is equipped with a single GPU, and that each pair of nodes has direct bidirectional connections [56]. The theoretical communication time is defined as the transfer time of messages, L/b, where L is the message size and b is the network bandwidth. For this analysis, we adopt the COO sparse format. For simplicity, we assume that the tensor on each GPU has identical d_G and the average densification ratio for all GPUs is γ_G^n. Additionally, the number of GPUs, n, is a power of 2.

• **Balanced Parallelism.** Communication in this scheme involves two steps. The traffic volume each GPU receives is $\frac{2(n-1)}{n} d_G M$ in the first step and $\frac{2(n-1)}{n}\gamma_G^n d_G M$ in the second step. The total traffic volume that each GPU receives is $\frac{2(n-1)}{n}(\gamma_G^n+1)d_G M$. resulting in a communication time of:

$$T_{bp} = \frac{n-1}{n}(\gamma_G^n+1)2Md_G/b. \quad (1)$$

• **Hierarchical Centralization.** Communication in this scheme is performed in $log n$ stages. In the i_{th} stage, the traffic volume that each GPU receives is $2\gamma_G^{2^{i-1}} d_G M$. The total traffic volume that each GPU receives is $2\sum_{i=1}^{\log n}\gamma_G^{2^{i-1}} d_G M$. The corresponding communication time is:

$$T_{sm} = \sum_{i=1}^{\log n}\gamma_G^{2^{i-1}} 2Md_G/b. \quad (2)$$

Balanced Parallelism as the practical optimal scheme. According to Theorem 1, the optimal scheme is determined by comparing Equations (1) and (2). Based on the characteristics of sparse tensors in DL models, we observe that Balanced Parallelism is often better than the other scheme because $\sum_{i=1}^{\log n}\gamma_G^{2^{i-1}} > \frac{n-1}{n}(\gamma_G^n+1)$ in practical distributed training. To illustrate, we consider two extreme cases. The first case is

Table 3: Comparison of different synchronization schemes for sparse tensors based on their dimensions.

Schemes	Communication	Aggregation	Partition	Balance	Note
AGsparse [35]	Ring, Hierarchy, Point-to-point	One-shot	Centralization	N/A	Cannot fully use overlaps
SparCML [56]	Hierarchy	Incremental	Centralization	N/A	to reduce traffic.
OmniReduce [23]	Point-to-point	One-shot	Parallelism	Imbalanced	Imbalanced communications
Balanced Parallelism	Point-to-point	Incremental	Parallelism	Balanced	The optimal scheme we find.

Table 4: The symbols used in this paper.

Symbols	Description
n	The number of GPUs
b	The network bandwidth
G	The dense tensor
M	The size of the dense tensor
d_G	The density of G
γ_G^k	The densification ratio of G with k GPUs

that any two tensors are *fully overlapped*, i.e., $\gamma_G^n = d_G$. The left-hand term becomes $\log n$, while the right-hand term is $2(n-1)/n < 2$, which is much smaller. For typically large n, such as $n \geq 16$, the right-hand term is several times smaller. The second case is that any two tensors have *no overlaps*, i.e., $\gamma_G^k = kd_G$. The left-hand term becomes $n-1$, while the right-hand term is $(n-1)(1+1/n)$, which is slightly larger than $n-1$. However, practical overlap ratios are much higher than 0.05, as shown in Figure 2a. More specifically, when $n = 16$, $\sum_{i=1}^{\log n} \gamma_G^{2^{i-1}} > \frac{n-1}{n}(\gamma_G^n + 1)$ even if the overlap ratio of any two tensors is as low as 0.05.

2.4 A Case Study on NMT model

In this section, we will empirically validate Theorem 1. As listed in Table 3, several synchronization schemes for sparse tensors are proposed [23, 35, 56]. We will compare their performance from an algorithmic perspective, i.e., only consider their theoretical communication time and ignore other overheads, such as the computation time for aggregations and the sparse tensor encoding and decoding overheads.

• **AGsparse [35].** It adopts [One-shot aggregation, Centralization], and separately collects non-zero gradients and the corresponding indices. It cannot fully exploit the overlaps among the sparse tensors to reduce traffic volume. Note that there are different implementations for AGsparse with different communication patterns [66].

• **SparCML [56].** It adopts Hierarchical Centralization. It cannot fully leverage the overlaps among the sparse tensors to reduce traffic volume. The performance of AGsparse and SparCML both depends on the overlaps. The fewer overlaps, the closer their performance is to optimal. However, as shown in Figure 2, sparse tensors across GPUs in DL models have significant overlaps.

• **OmniReduce [23].** It adopts [Point-to-point, One-shot aggregation, and Parallelism]. OmniReduce consists of workers and aggregators. It splits a gradient tensor into blocks of gradients and only sends non-zero blocks, i.e., blocks with at least one non-zero gradient, to aggregators for aggregations. OmniReduce does not need to transmit indices for non-zero gradients. It requires multiple aggregators for better scalability. However, its performance suffers from imbalanced communications because it evenly partitions tensors.

The sparse data formats are as proposed by each scheme, i.e., AGsparse and SparCML use COO; OmniReduce uses tensor block format. We assume the sparse data format of Balanced Parallelism to be COO for a fair comparison.

The case where Balanced Parallelism is optimal. Figure 8 compares the performance of these synchronization schemes to synchronize sparse tensors in NMT with a batch size of 64. The comparison with other DL models has similar trends. Their communication times are normalized to *AllReduce*, which is the synchronization scheme for dense tensors. The communication time of AGsparse increases linearly with the number of GPUs. It performs worse than AllReduce with more than 40 GPUs because it does not leverage the overlaps among the sparse tensors to reduce traffic volume. OmniReduce outperforms AllReduce with a small number of GPUs. However, its performance improvement becomes marginal with more than 64 GPUs. Due to the skewed distribution of non-zero gradients, most of the non-zero gradients are in one partition, leading to imbalanced communications. In addition, the tensors become denser after aggregation. When splitting this partition into tensor blocks (e.g., each block has 256 gradients [23]), most of them are non-zero blocks. Therefore, almost all gradients in this partition are sent to one aggregator, and it becomes the communication bottleneck. SparCML is worse than AllReduce with a large number of GPUs due to the duplicated indices and their gradients received at each GPU. In contrast, Balanced Parallelism greatly outperforms existing synchronization schemes and AllReduce. For example, existing schemes cannot reduce communication time compared to AllReduce with 128 GPUs, but the communication time of Balanced Parallelism is still 36% lower than AllReduce.

The case where SparCML is optimal. This case is rare in practical distributed training. We have to reduce the batch size of NMT to a very small value for demonstration. For example, when the batch size is 1 and the number of GPUs is 8, the density of the gradient tensors of the embedding layer is less than 0.1%, and the sparse tensors hardly have overlaps, i.e., $\gamma_G^k \approx k$. In this scenario, SparCML outperforms AGsparse

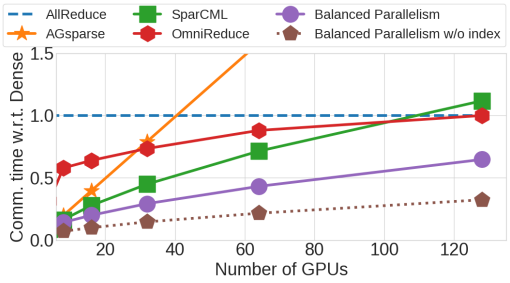


Figure 8: Comparison of different schemes for synchronizing sparse tensors in NMT [40] with a batch size of 64.

and Balanced Parallelism by 4% and 9%, respectively. As the GPU number increases, SparCML still outperforms Balanced Parallelism, but the performance gap becomes more marginal.

The index communication overhead is non-negligible. When the sparse data format is COO, each non-zero gradient in a sparse tensor comes with an index, significantly inflating the traffic volume for gradient synchronization and communication time. To demonstrate its costly overhead, Figure 8 displays an ideal scheme named Balanced Parallelism without index, in which only non-zero gradients are synchronized with Balance Parallelism without any index information. Compared to the ideal scheme, Balanced Parallelism doubles the communication time.

3 ZEN System

We crystalize the above observations and insights into a holistic gradient synchronization system, ZEN (Figure 1), which leverages the sparsity in DL models to minimize the synchronization time in distributed training. ZEN comprises both schemes described in Theorem 1 to minimize communication time under different scenarios. At the runtime, ZEN collects the lightweight sparsity profiling results of the first few iterations and intelligently determines the optimal scheme by comparing Equations (1) and (2).

We will focus on Balanced Parallelism in this section because to date no existing solution realizes it, while SparCML [56] has been proposed. We first formulate the problem for Balanced Parallelism (§3.1) and then develop a hierarchical hashing algorithm to solve it (§3.2). We then design a new sparse data format to minimize the communication overhead incurred by gradient indices (§3.3) in sparse tensors.

3.1 Balanced Parallelism Formulation

For convenience, we borrow the concepts of workers and servers from the Parameter Servers (PS) architecture [34] to Balanced Parallelism. Because there are two steps in Balanced Parallelism for gradient synchronization, we also call them Push and Pull, respectively.

Suppose that there are n workers and n servers in Balanced Parallelism. $I_i \subset \mathbb{N}_+$ is the indices of non-zero gradients generated by worker i. We define the problem of achieving balanced parallelism as follows.

Problem 1. *Let I denote the union of $\{I_1, I_2, \cdots, I_n\}$. We would like to have a mapping $f : I \to [n]$ such that:*

1. *For every $i \in [n]$ and $j \in [n]$, the cardinality of set $\{idx \in I_i | f(idx) = j\}$ is equal to $|I_i|/n$.*
2. *For every $j \in [n]$, the cardinality of set $\{idx \in I | f(idx) = j\}$ is equal to $|I|/n$.*

Here we elaborate more on the two requirements for the mapping f accordingly as follows.

1. **Load balance in Push.** For every worker, mapping f needs to decompose its non-zero gradients evenly into n partitions. Therefore, workers can transmit the same amount of non-zero gradients to each server.
2. **Load balance in Pull.** Each of the servers should have the same number of non-zero gradients after aggregation. It also implies that the same index from different workers should be sent to the same server.

Definition 8 (The imbalance ratio)**.** *Given a mapping f that decomposes I_i into n partitions, denoted as $\{I_i^1, \cdots, I_i^n\}$. The imbalance ratio of Push with f is $\max_{i,j\in[n]}\{n|I_i^j|/|I_i|\}$.*

Let I denote the union of $\{I_1, I_2, \cdots, I_n\}$ and the sets of indices on the n servers after aggregation are $\{\mathbb{I}_1, \mathbb{I}_2, \cdots, \mathbb{I}_n\}$. The imbalance ratio of Pull with f is $\max_{i\in[n]}\{n|\mathbb{I}_i|/|I|\}$.

Based on Definition 8, the imbalance ratio of Push and Pull in Balanced Parallelism is 1. Our goal is to minimize the imbalance ratio for any distributions of non-zero gradients.

Data-dependent solutions cause costly overheads. Due to different sets of indices on different workers, data-dependent solutions need to analyze their overall distribution and calculate one mapping for all workers, inevitably incurring non-negligible computation overheads. In our testbed, the measured computation cost is orders of magnitude greater than the iteration time. Hence, we cannot afford to apply a data-dependent algorithm and obtain a mapping f for every iteration. A possible approach is to compute f periodically and maintain it for the next iterations. However, this approach still leads to high imbalance ratios due to the varying index distributions across iterations. One strawman following this approach is to sort the index set I, evenly partition it into n parts, and use the boundary indices as thresholds to partition the index sets in the next iterations. When computing the thresholds every 1000 iterations for the NMT model [40] with $n = 16$ and applying these thresholds to the following iterations, the imbalance ratio of Push fluctuates between 1.4 and 5.1, causing imbalanced communications between servers. Moreover, the imbalanced communications introduced by data-dependent solutions make it difficult to estimate the iteration time. Many resource scheduling mechanisms for GPU

clusters assume predictable and stable iteration times to allocate resources to training jobs [41, 48, 54]. It is cumbersome to schedule GPU resources with fluctuating communication time.

3.2 Data-independent Hierarchical Hashing

Due to the limitations of data-dependent solutions, we must develop a data-independent solution to achieve load balance in both Push and Pull with negligible computation overheads.

A naive solution to solve Problem 1 is to apply a universal hash function across multiple threads on a GPU, i.e., each thread independently operates hash functions on a disjoint input and writes them into a hash memory. Unfortunately, this approach is lossy. When two indices are hashed to the same location in a hash memory, only one index can be written into the hash memory while the other is overwritten, leading to significant information loss. For example, when the hash memory size equals the size of a dense tensor with 10% density, 17.5% gradients are lost due to hash collision. Increasing the hash memory size can reduce the information loss, but it introduces non-negligible computational overhead because the algorithm must extract the non-zero values from the hash memory after the hash operations. To illustrate, consider the embedding layer (1.6GB) in LSTM [44]. When the hash memory size is four times the tensor size, the information loss rate decreases to 4.8%. However, the extraction overhead increases to 41.8ms using the built-in `nonzero()` API in PyTorch 2.2 on an NVIDIA V100 GPU. This overhead is unacceptable, as it is roughly 40% of the single-GPU iteration time (114 ms in our testbed). In addition, increasing the hash memory size significantly increases GPU memory usage, potentially causing out-of-memory issues and crashing the training process.

We develop a novel hierarchical algorithm to solve Problem 1. We leverage multiple threads on GPUs to efficiently perform hash functions. We achieve no information loss and minimal GPU memory usage by using four techniques.

Technique #1: communication-oriented hash memory management. The hash memory is divided into multiple partitions and the data written into each partition is for the corresponding server. Each partition is further divided into parallel memory and serial memory to prevent data loss. When a thread within a GPU executes a hash function, it initially examines for hash collisions by checking if the hashed location is occupied. In the absence of a collision, the data are written to the parallel memory, enabling concurrent writing operations across all threads. In case of a collision, the colliding indices are sequentially written to the serial memory using an atomic operation, allowing only one thread to write data at a time. Although the indices written in the memory are in random order, there is no need to sort them because their orders do not affect the aggregated results.

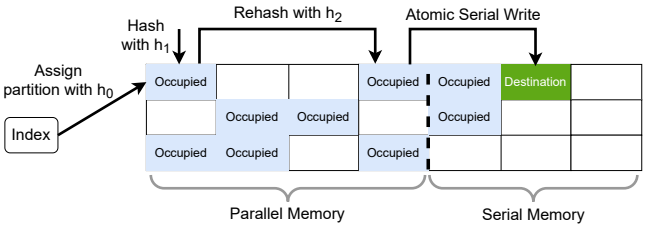


Figure 9: Demonstration of the hierarchical hashing algorithm. We perform parallel hashing on the indices. For each index, we use hash function h_0 to assign its partition. We next use hash function h_1 to assign it to the first location. However, because this location is occupied, we rehash it with function h_2 to the fourth location. As it is also occupied, we serially write the index into the serial memory with an atomic operation.

Unfortunately, we observe that the expense of serial writing becomes significant when the hash collision rate is elevated.

Technique #2: multiple hash functions for each thread in GPU. Multiple hash functions are used to reduce the cost of serial writing. When a hash collision occurs, a GPU thread can rehash the index with a new hash function to another location in the parallel memory. There is a chance that this new location is available and the number of sequential write operations to the serial memory can be reduced. Although hash collision still exists even with multiple hash functions, we observe that the collision rate is less than 1% with four hash functions and the cost of serial writing into the serial memory becomes acceptable.

However, rehashing with multiple hash functions can cause incomplete aggregations. Because different GPUs have different sets of indices for non-zero gradients, their sequences of indices being hashed are also different. Therefore, the location of a particular index can be different between GPUs. For example, two indices idx_1 and idx_2, where $idx_1 < idx_2$, are hashed to the same location with the first hash function. GPU 1 has idx_2; GPU 2 has both idx_1 and idx_2. In GPU 1, the location of idx_2 is determined by the first hash function, but in GPU 2, the location of idx_2 is determined by the second hash function because the location hashed by the first one has been occupied by idx_1. Subsequently, partitioning the memory will lead to the same index assigned to different partitions on different GPUs, resulting in incomplete aggregations.

Technique #3: consistent hierarchical hashing across workers. We propose a two-level hierarchical hashing algorithm to guarantee complete aggregations. The first-level hashing determines the partition to which an index belongs and guarantees that an index will belong to the same partition across all GPUs. The second-level hashing determines its locations in this partition. To ensure that the same index from different workers can be sent to the same server, ZEN allocates the same first-level hashing function to all workers but allows

them to have independent second-level hashing functions.

Technique #4: lock-free read-after-write mechanism. There is a concern that two values from two threads can be hashed to the same available location simultaneously, and this hash collision cannot be detected by conventional mechanisms, leading to information loss. ZEN uses a read-after-write mechanism to check this collision, eliminating this type of information loss. After memory writing, a thread reads the value stored in the location and this operation has no dependency on the values from other threads. If the value equals what it writes, this thread will move on to the next input. However, if the value is not what it writes, it implies that there is a hash collision and its value is overwritten. Then the thread will take a rehash or serial writing.

Hierarchical hashing algorithm. The pseudocode with the four techniques is shown in Algorithm 1. An example of this algorithm is also illustrated in Figure 9. Given a dense tensor G and the indices of its non-zero gradients I, it allocates a memory x with shape $n \times (r_1 + r_2)$, where n is the number of partitions, r_1 is the memory size for parallel hashing operations, and r_2 is the serial memory size. It performs a hashing operation for every $idx \in I$ in parallel (Lines 4-17). A universal hash function $h_0 : \mathbb{N}_+ \to [n]$ is used to locate idx to partition $p = h_0(idx)$ (Line 5). The algorithm also needs k universal hash functions $H = \{h_1, \cdots, h_k\}$ with $h_i : \mathbb{N}_+ \to [r_1]$. After determining the partition p, the algorithm attempts to find an available destination $x[p][h_1(idx)]$ with h_1. If this location is available, idx is written into it. Otherwise, the algorithm rehashes idx with h_2 to find a new location. It rehashes an index for at most k rounds and uses h_i as the hash function for round i until it finds an available destination (Lines 6-16). The algorithm writes idx to the serial memory of partition p after k rehash attempts fail (Lines 8-11). Serial writing is an atomic operation (Lines 9-10) to ensure no information loss. Once all indices are written into the memory, it extracts sparse tensors from the memory (Lines 19-23).

Next, we will analyze the properties of Algorithm 1 and its imbalance ratio.

• **No impact on model accuracy per iteration.** ZEN is an efficient communication scheme for sparse tensor synchronization and it has no impact on the iteration-wise convergence rates of models. For distributed training with natural tensor sparsity, the aggregated result of sparse tensors with ZEN is identical to that of the corresponding dense tensors with AllReduce. When using sparsification algorithms, some accuracy loss may occur due to the compression itself, depending on the sparsity level. However, ZEN does not introduce additional accuracy loss beyond what is inherent to the sparsification algorithm. Specifically, ZEN's Push ensures no further information degradation through its four techniques, and both its gradient aggregation and Pull operations are lossless. The aggregated tensor after sparse synchronization with ZEN is identical to that produced by other schemes such as AGsparse [35] and OmniReduce [23]. Consequently, ZEN preserves the same model accuracy per iteration while achieving shorter wall-clock iteration time. More detailed accuracy evaluations and discussion are shown in Figure 18.

Algorithm 1: Hierarchical Hashing Algorithm

Input: G is a dense tensor and $I \subset \mathbb{N}_+$ is the indices of its non-zero gradients. n is the number of partitions. Each partition has a memory size $r_1 + r_2$, where r_1 and r_2 are the memory sizes for parallel and serial operations, respectively. $h_0 : \mathbb{N}_+ \to [n]$ is a universal hash function. $H = \{h_1, \cdots, h_k\}$ is a set of universal hash functions where $h_i : \mathbb{N}_+ \to [r]$.

Output: The partitioned sparse tensors.

```
1  Function hierarchical_hash(I, G, h_0, H):
2      Allocate memory x ← 0^{n×(r_1+r_2)}
3      Allocate atomic counters c ← 1^n
4      foreach idx ∈ I in parallel do
5          p ← h_0(idx)
6          for i ← 1 to k+1 do
7              q ← h_i(idx)
8              if i = k+1 then
9                  q ← atomicAdd(c[p], 1)
10                 x[p][q] ← idx
11             end
12             if x[p][q] == 0 then
13                 x[p][q] ← idx
14                 break
15             end
16         end
17     end
18     output = []
19     for i ← 0 to n−1 do
20         indices = nonzero(x[i])
21         values = G[indices]
22         output.append((indices, values))
23     end
24     return output
```

• **Negligible memory and extraction overhead.** Thanks to multiple hashing functions and serial memory, Algorithm 1 can achieve no information loss with a small memory size, which is typically less than 150MB (<1% GPU memory capacity) in our implementation for all the evaluated models. The incurred overhead to extract the indices from the memory after hashing (Line 20 in Algorithm 1) becomes negligible.

• **No dependency on workloads.** Note that we impose no assumptions on the data distributions and only use the property of universal hashing algorithms. Hence, Algorithm 1 can obtain a general theoretical guarantee for different distributions of non-zero gradients in DL training workloads.

• **Guaranteed imbalance ratio.** Because the hash function h_0 determines the partition of each index, the imbalance ratio of Algorithm 1 is guaranteed by the following theorem.

Theorem 2 (Load Balance of Algorithm 1)**.** *Given a dense*

tensor G with $|G|$ parameters. Algorithm 1 provides a mapping $f : I \to [n]$ such that

1. *With probability at least $1-1/n$, its imbalance ratio of Push is at most $1+\Theta(\sqrt{\frac{n\log n}{|G|d_G}})$.*
2. *With probability at least $1-1/n$, the imbalance ratio of Pull is at most $1+\Theta(\sqrt{\frac{n\log n}{|G|d_G^n}})$.*

The proof can be found in Appendix B.2. Because $n\log n$ is orders of magnitude smaller than $|G|$, Algorithm 1 performs a good approximation to the exact solution of Problem 1 for both Push and Pull. Suppose $n=128$, $|G|=10^7$, and $d_G^n=0.5$, $\sqrt{\frac{n\log n}{|G|d_G^n}} < 0.02$. Its practical imbalance ratio is always less than 1.1 for the six models studied.

3.3 Cost-Efficient Encoding Scheme

3.3.1 Existing Sparse Formats are Inefficient

There are several sparse formats to represent sparse tensors for their synchronization. Unfortunately, none of them can minimize the overhead incurred by the indices for non-zero gradients. We assume that the data type of gradients is FP32.

• **COO.** It is efficient with low tensor density [39, 74]. However, it doubles traffic volume and becomes inefficient for a high density. As shown in Figure 2b, the tensors become denser after aggregation. For example, the average tensor density of BERT increases from 1.06% to 40.8% after the aggregation of the sparse tensors from 128 GPUs. Theoretically, transmitting sparse tensors in Pull can reduce the traffic volume by 2.5× compared to transmitting dense tensors, but the reduction shrinks to only 1.2× due to the indices for non-zero gradients.

• **Tensor block.** It is used in OmniReduce [23]. A dense tensor is split into blocks of gradients, and only non-zero blocks are transmitted. However, it is inefficient when the tensor density is high. When splitting a tensor with high density into tensor blocks (e.g., each block has 256 gradients), most of them have at least one non-zero gradient and become non-zero tensor blocks. The synchronization scheme has to communicate a large amount of zero gradients. We evaluated the percentage of gradients communicated with the tensor block format compared to the tensor density. DGC with top 5% gradients is applied to OPT-2.7B. When the block size is 256, more than 30% gradients are communicated, much higher than the tensor density of 5%.

• **Bitmap.** It only needs one bit to indicate whether a gradient is zero or not. Unfortunately, a conventional bitmap still incurs non-negligible traffic volume to identify non-zero gradients. When the dense tensor G is evenly partitioned, the indices of non-zero gradients in each server are in a subrange of $[1,|G|]$, where $|G|$ is the number of gradients in G. For example, when $|G|=15$ and there are three servers, the index range in Server i is $[5i+1, 5(i+1)]$. The extra bitmap size required to represent the indices of non-zero gradients in each server is $|G|/n/32$ when the data type of gradients is FP32. The total bitmap size received by each worker is $|G|/32$. When the tensor size of G is 856MB, which equals the embedding table size in DeepFM, the total bitmap size is 27 MB. Although Algorithm 1 enables load balance, the non-zero gradients in each server are randomly distributed throughout the range. If we still use a bitmap to represent the indices, the extra bitmap size in each server is $|G|/32$ and the total bitmap size received at each worker becomes $n|G|/32$, which linearly increases with the number of servers. When there are 16 servers, the total bitmap size is 428MB.

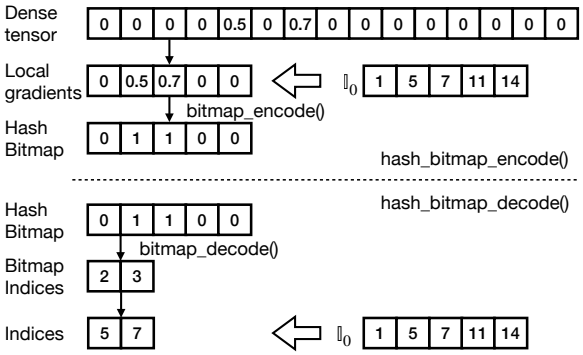


Figure 10: An illustration of the hash bitmap.

Algorithm 2: The hash bitmap

Input: G is a dense tensor. $\mathbb{I}_i = \{idx \in [1,|G|] \mid h_0(idx) = i\}$, where h_0 is defined in Algorithm 1.

```
Function hash_bitmap_encode(G, I_i):
    local_gradients = G[I_i]
    hash_bitmap = bitmap_encode(local_gradients)
    return hash_bitmap
Function hash_bitmap_decode(I_i, hash_bitmap):
    bitmap_indices = bitmap_decode(hash_bitmap)
    indices = I_i[bitmap_indices]
    return indices
```

3.3.2 Hash Bitmap

We develop a novel *hash bitmap* for ZEN to minimize the overhead to represent indices for non-zero gradients in Pull. Given a dense tensor G and h_0 in Algorithm 1, the set of indices $\mathbb{I}_i = \{idx \in [1,|G|] \mid h_0(idx) = i\}$ in each worker that should be pushed to Server i is determined, although it is not in a continuous range. Since $\mathbb{I}_i$ and $\mathbb{I}_j$ are disjoint when $i \neq j$, it provides an opportunity to construct the bitmap based on $\mathbb{I}_i$, rather than the entire range.

Figure 10 illustrates how the hash bitmap works for $\mathbb{I}_0$ with $|G|=15$ and three servers. The indices for the two non-zero gradients are $\{5,7\}$. `hash_bitmap_encode()` is used to encode the indices. Given a dense tensor G, it first extracts local gradients according to the indices in $\mathbb{I}_0$. It then

encodes the local gradients into a bitmap. Because the second and third gradients are non-zero, the second bit and the third bit in the hash bitmap are 1 and the other bits are 0. `hash_bitmap_decode()` is used to decode the hash bitmap to a set of indices. It first decodes a hash bitmap to the bitmap indices, which are the indices of 1. For example, because the second and third bits in the hash bitmap are 1, the bitmap indices are $\{2,3\}$. It then uses the bitmap indices as the indices to extract the corresponding values in $\mathbb{I}_0$ as the global indices for non-zero gradients. In this example, the values are $\{5,7\}$, which are exactly the indices for the two non-zero gradients. The pseudocode is shown in Algorithm 2.

The function `hash_bitmap_encode()` is invoked on each server, which then broadcasts the hash bitmap to all workers. After each worker receives the hash bitmaps from all the servers, it invokes `hash_bitmap_decode()` to decode the hash bitmaps to the indices with the corresponding $\mathbb{I}_i$. Note that $\mathbb{I}_i$ is computed and sorted offline and it remains unchanged for the same h_0 on both servers and workers.

The hash bitmap guarantees that the total hash bitmap size received at each worker is constantly $|G|/32$ in ZEN's Pull. Suppose that there are n servers. The set of indices that should be pushed to Server i is $\mathbb{I}_i = \{idx \in [1,|G|] \mid h_0(idx) = i\}$. With `hash_bitmap_encode()`, the size of the hash bitmap encoded in Server i is $|\mathbb{I}_i|/32$. Because each worker needs to receive the hash bitmap from all servers, the total size is $\sum_{i=0}^{n-1} |\mathbb{I}_i|/32 = |G|/32$. Zen still uses COO to represent sparse tensors in Push due to the low tensor density.

4 Implementation

We implement ZEN [1] with about 900 lines of Python code, 250 lines of CUDA code, and 500 lines of code for hacking ColossalAI [36]. For DL model training, we use ColossalAI to implement data parallelism and hybrid parallelism (data parallelism + tensor parallelism, with tensor parallelism degree set to 8, typically within a single node). We also extend ColossalAI to support the Gemma2 [65] model from Google with custom policies for efficient tensor parallelism.

For natural tensor sparsity, we only apply sparse tensor synchronization schemes to gradients from the embedding layers for their communications across machines. For the sparsification algorithm, we use DGC [39] and adapt it to tensor parallelism: sampling data locally on each device, gathering these samples across all devices, computing a global top-k threshold based on aggregated data, and applying this threshold locally to determine top-k values on each device. We register a `custom_comm_hook` in PyTorch's DDP [35] to enable DGC and our synchronization scheme. Gradient computations overlap with gradient synchronization during backward propagation. Gradient tensors are fused into communication buckets in DDP before applying DGC to tensors [73]. We determined a 128MB bucket size that works best for the evaluated models.

We implement the hierarchical hash algorithm in CUDA C and use it as an extension for PyTorch. The used hash function is MurmurHash [9]. We set different seeds for MurmurHash to generate different hash functions. At the beginning of training, ZEN generates a set of random seeds and broadcasts them to all GPUs to ensure hash consistency among workers. ZEN communicates dense tensors within machines with ReduceScatter/AllGather [25, 66] because NVLink is typically equipped in GPU machines. In our evaluations, Balanced Parallelism is the optimal scheme for the studied models.

5 Evaluation

5.1 Experimental Setup

Testbeds. We use 16 p3.16xlarge instances from AWS EC2 for our evaluations. Each machine has 8 NVIDIA V100 GPUs (16GB GPU memory) and NVLink is equipped to support intra-instance GPU-to-GPU communications. These instances are connected by a 25 Gbps network. We also evaluate end-to-end training throughput with 16 p3dn.24xlarge from AWS EC2. Each machine has 8 V100 GPUs (32GB GPU memory) and NVLink. These instances are connected by a 100 Gbps RDMA network with AWS's Elastic Fabric Adapter (EFA). Each machine runs Ubuntu 20.04 and the software environment includes CUDA-11.0, PyTorch-2.2, NCCL-2.7.8, CuPy-11.0, and ColossalAI-v0.4.6. Each p3dn.24xlarge instance also includes aws-ofi-nccl-v1.9.2 and libfabric-v1.11.1 to support EFA.

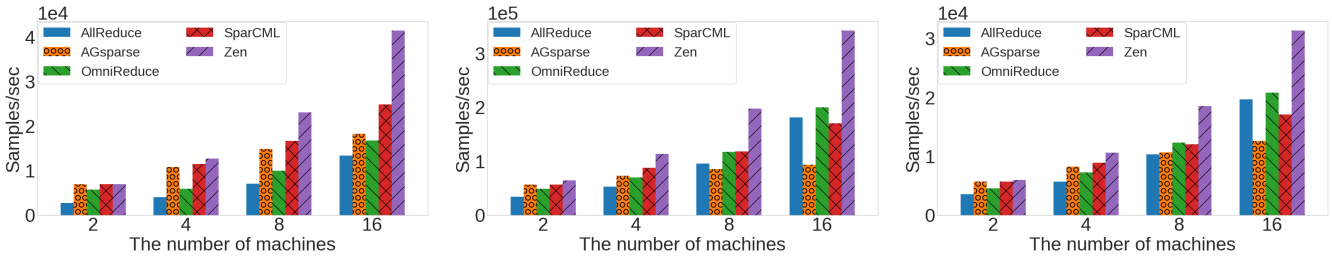
Workloads. For natural tensor sparsity evaluation, we use three models, LSTM [44], DeepFM [26], and NMT [40], as listed in Table 1. The datasets are One Billion Word [1], Criteo [2], and IWSLT 2014 De-En [3], respectively. The batch sizes are 128, 1024, and 64, respectively. These models can fit in the memory of each GPU. The per-GPU batch size is kept constant as the number of GPUs increases. For the tensor sparsity from sparsification algorithms, we use three LLMs, Llama3.2-3B, OPT-2.7B, and Gemma2-2B, as listed in Table 2. We use RedPajama [75] as the training corpus. Because these models cannot fit into a single GPU, we use tensor parallelism (TP) with degree 8. The batch size within an TP group is 1 and the sequence length is 1024 tokens.

Baselines. We compare ZEN with AGsparse [35], SparCML (SSAR_Recursive_double) [56], and OmniReduce [23]. We use `AllReduce` [59] for the synchronization of dense tensors. We apply DGC [39] to the three LLMs and select the top 5% gradients from their gradient tensors.

5.2 DL Training Evaluation

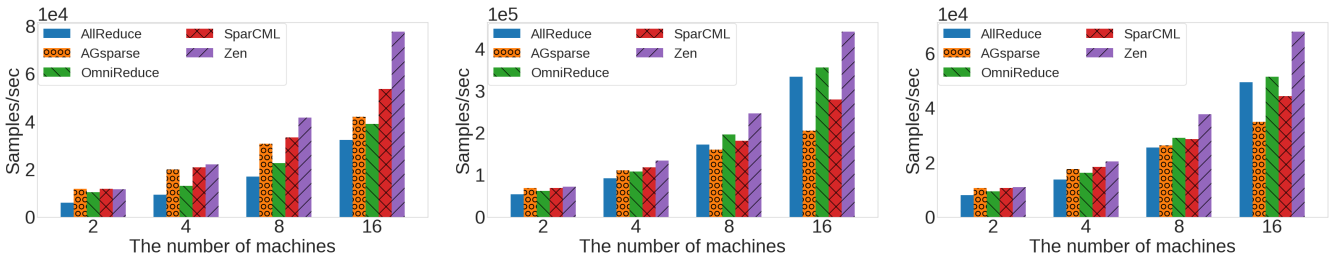
In this section, we present the training efficiency of ZEN on the six models and compare it with the baselines. We set $k = 3$,

[1] https://github.com/zhuangwang93/ZEN



(a) LSTM (b) DeepFM (c) NMT

Figure 11: Training throughput of models with natural tensor sparsity in a 25 Gbps network.



(a) LSTM (b) DeepFM (c) NMT

Figure 12: Training throughput of models with natural tensor sparsity in a 100 Gbps RDMA network.

$r_1 = 2|G|d_G$, and $r_2 = r_1/10$ for Algorithm 1.

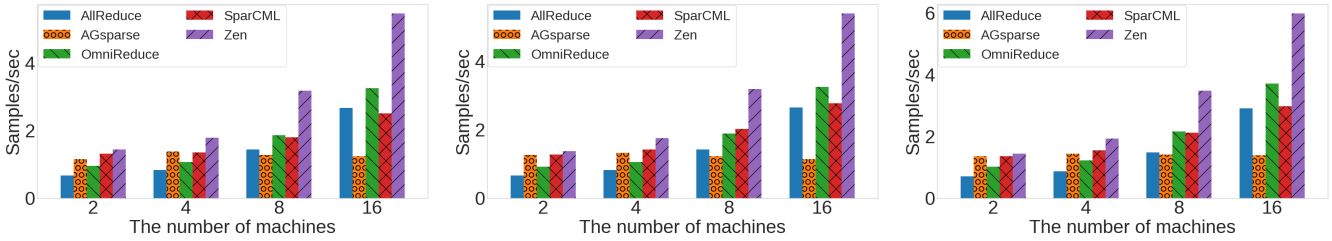
Training throughput for models with natural sparsity. Figure 11 shows the training throughput of models with natural tensor sparsity in a 25G network. ZEN outperforms all baselines by processing more samples in a second. When training LSTM with 16 machines, ZEN achieves up to a 1.67× speedup over SparCML (the best baseline), a 2.48× speedup over OmniReduce and a 3.1× speedup over AllReduce. In both DeepFM and NMT, the best baseline is OmniReduce. ZEN achieves 1.44× speedup and 1.51× speedup over OmniReduce for DeepFM and NMT, respectively. As the number of machines increases, the benefits of ZEN over SparCML and OmniReduce are enlarged, indicating ZEN's great scalability. When increasing the network bandwidth from 25 Gbps to 100 Gbps, ZEN still has noticeable end-to-end speedups, as shown in Figure 12. Specifically, in LSTM, ZEN achieves up to 1.44× speedup over SparCML, which is the best baseline. In DeepFM and NMT, ZEN achieves up to 1.33× and 1.38× speedups over AllReduce and up to 1.25× and 1.32× speedups over the best baseline OmniReduce.

Training throughput for models with gradient compression. ZEN has great speedups over the baselines for gradient compression scenarios, as shown in Figure 13. For Llama3.2-3B, it achieves up to a 1.68× speedup over OmniReduce, a 2.19× speedup over SparCML, and a 2.02× speedup over AllReduce. In OPT-2.7B and Gemma2-2B, ZEN achieves up to 2.10× and 2.04× speedups over AllReduce, and 1.66× and 1.61× speedups over OmniReduce. We also evaluate the training throughput improvement in a 100 Gbps RDMA network, as shown in Figure 14. ZEN outperforms AllReduce by 64%, 45%, and 44% in Llama3.2-3B, OPT-2.7B, and Gemma2-2B, respectively. OmniReduce is the best baseline for the three models, while ZEN still achieves up to 1.32× in Llama3.2-3B, 1.31× in OPT-2.7B, and 1.27× in Gemma2-2B over OmniReduce. These results demonstrate ZEN fully leverages sparsity in DNN models to optimize training efficiency.

Communication improvement. The performance of ZEN is driven by the reduction in communication time. Figure 15 shows the speedups of different synchronization schemes over AllReduce with 16 machines in a 25 Gbps network. The speedup of OmniReduce is up to 1.58×. We observe that the performance of AGsparse, Sparse PS, and SparCML can be even worse than AllReduce in some cases. They use COO as the sparse format. With high density, the sparse tensor size with COO is larger than the dense tensor size. The communication time of AGsparse increases linearly with the number of machines. With SparCML, the overlaps among the sparse tensors can be received multiple times on each GPU. In contrast, ZEN achieves 6.77× speedup for LSTM and 3.51× speedup for Gemma2-2B. Its speedups over SparCML and OmniReduce are up to 2.82× and 5.16×, respectively. ZEN also achieves 2.10× speedup for DeepFM and 3.44× speedup for OPT-2.7B over Allreduce.

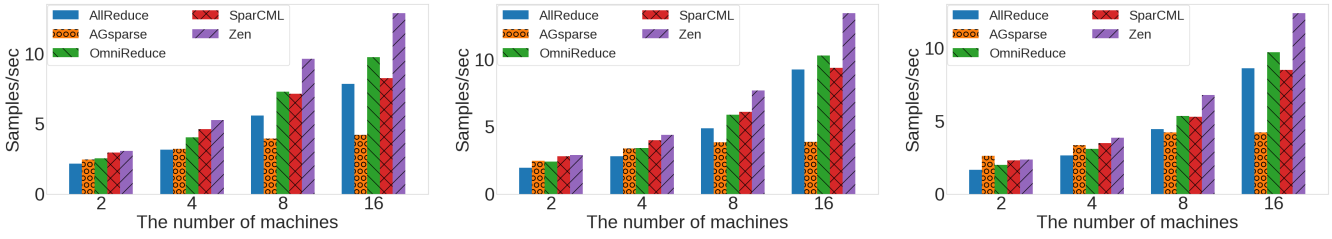
5.3 Understanding ZEN

A study on parameters for Algorithm 1. We simulate a tensor with 214M parameters (same as the embedding gradients in DeepFM) and vary its density to perform a study on both r_1 and k in Algorithm 1. We first study the parameter r_1. We



(a) Llama3.2-3B (b) OPT-2.7B (c) Gemma2-2B

Figure 13: Training throughput of models with tensor sparsity from sparsification algorithms in a 25 Gbps network.



(a) Llama3.2-3B (b) OPT-2.7B (c) Gemma2-2B

Figure 14: Training throughput of models with tensor sparsity from sparsification algorithms in a 100 Gbps RDMA network.

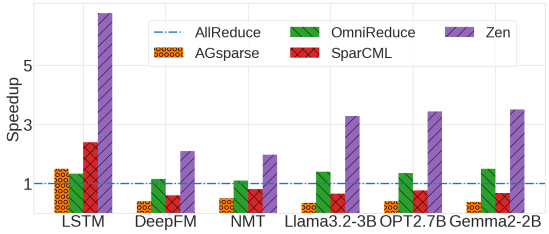
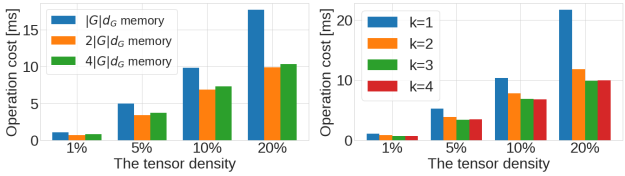


Figure 15: Communication speedups over AllReduce in a 25 Gbps network.



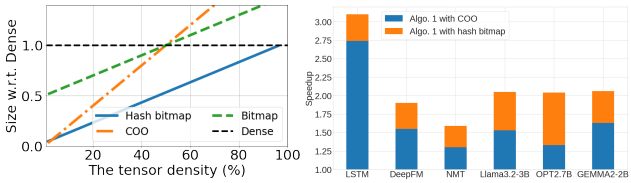
(a) (b)

Figure 16: The computation overhead of Algorithm 1. (a) Different memory sizes and (b) Different numbers of rehash.

set $r_2 = r_1/10$ and $k = 3$. As shown in Figure 16a, when we increase r_1 from $|G|d_G$ to $2|G|d_G$, there is a notable reduction in operation cost because the larger memory size increases the probability of successful parallel writing and reduces the workload of serial writing. But when we further increase r_1 from $2|G|d_G$ to $4|G|d_G$, it leads to a longer computation time. There are two reasons behind this effect. Firstly, the workload of serial writing is already low with $2|G|d_G$ memory. Further increasing r_1 only marginally reduces the computation overhead. Secondly, a larger memory size can increase the computation overhead to extract the indices (see Algorithm 1) and thus degrade the overall performance. Figure 16b shows the computation costs versus k when we use $2|G|d_G$ memory. Increasing k from 1 to 3 can reduce the operation cost as it alleviates serial writing workload, but $k = 3$ and $k = 4$ have very similar operation costs.

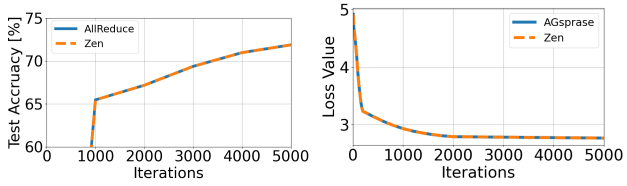
Negligible compute overhead. ZEN's performance gains mainly come from communication savings by only synchronizing non-zero values in gradient tensors and load-balancing the traffic volume across GPUs at the cost of additional hashing computation, which is much smaller than the communication savings. We use the tensor with 214M parameters in DeepFM for example. The hashing computation overhead is around 6 ms in our testbed as the practical tensor density is 5.2% after intra-machine aggregation. When the network bandwidth is 25 Gbps, the communication saving over AllReduce is around 270 ms, which suggests that ZEN's compute overhead is negligible. Even with a 100 Gbps RDMA network, the hashing compute time only accounts for 9% of communication savings.

Hash Bitmap. We show the effectiveness of hash bitmap in representing the indices of non-zero gradients. Figure 17a shows the tensor size with different sparse formats. The sizes are normalized to the dense tensor and there are 16 servers. The tensor density is the total density of all servers after aggregation. The gap between the hash bitmap and the COO increases with the tensor density. It also significantly outperforms the bitmap. In addition, the hash bitmap can still reduce traffic volume with a density of 95% compared to the dense



(a) (b)

Figure 17: (a) The effectiveness of the hash bitmap; (b) The performance breakdown of ZEN with 16 machines in a 25 Gbps network.



(a) DeepFM (b) OPT-2.7B

Figure 18: ZEN has no impact on model accuracy per iteration.

tensor, but the bitmap and the COO cannot save the volume when the density is greater than 50%. The performance of tensor blocks varies with the distribution of non-zero gradients. Some sparse tensors in the studied DL models can even transmit higher traffic volume than COO because a non-zero block has more zero gradients than non-zero gradients.

Performance breakdown. We break down the performance of ZEN by Algorithm 1 and the hash bitmap. Figure 17b illustrates the training throughput speedup breakdown over AllReduce with 16 machines. It can be seen that the primary performance benefits of ZEN come from Algorithm 1, with the hash bitmap format providing noticeable additional benefits. For example, when the data format is COO after applying Algorithm 1, the speedup is 2.74× and 1.53× for LSTM and Llama3.2-3B, respectively. Replacing COO with the hash bitmap can further improve the speedups by 13% and 34%.

Model accuracy validation. We first validate that ZEN has no impact on model accuracy per iteration compared to training with AllReduce for natural sparse tensors. Figure 18a shows the model accuracy of DeepFM with 16 machines in a 25 Gbps network. The test accuracy per iteration with ZEN matches exactly that with AllReduce. This is because ZEN balances the synchronization traffic across workers to reduce the iteration time without information loss for gradient synchronization. We then evaluate ZEN's impact on model quality for training with sparsification algorithms. Figure 18b shows the training loss values of OPT-2.7B with DGC selecting top 5% gradients. AGsparse [35] is used as the baseline for sparse tensor synchronization, and we can see that ZEN and AGsparse have identical iteration-wise loss curves, demonstrating that ZEN retains the model convergence rates and model accuracy per iteration as other communication schemes.

6 Related Work

Related work on schemes to support sparse tensor synchronization has already been discussed in Section 2.4.

Acceleration of dense tensor synchronization. ATP [33] and SwitchML [57] exploit programmable switches for the synchronization of dense tensors. BytePS [29] uses the spare CPU and bandwidth resources in GPU clouds to optimize communications. These approaches disregard the values of the gradients and communicate all gradients. In contrast, Zen leverages sparsity in DNN models and only transmits non-zero gradients to reduce the synchronization time.

Acceleration of sparse tensor synchronization. Parallax [31] utilizes parameter servers for sparse tensor synchronization and it cannot achieve balanced communications in synchronization. ZEN can achieve balanced communications using a novel hierarchical hashing algorithm. Flare [21] and Libra [52] use programmable switches to accelerate sparse tensor communications, but they rely on specific hardware. Ok-Topk [38] proposes a novel sparse allreduce algorithm; however, its inherently data-dependent balancing strategy results in significant overhead. ZEN analyzes the characteristics of sparse tensors and explores the design space for communication schemes to determine the optimal one, but prior approaches did not consider these factors.

Hash algorithms for load balance. Previous works attempt to achieve load balance using hashing [11,13,14,19,76]. They typically assign two partitions to a given index using two hash functions and then select the partition with more available memory [20,45,46]. However, these methods require serial writing of indices to memory and cannot leverage multiple threads in GPUs. In contrast, ZEN enables parallelizable computing on GPUs. While DRAGONN [74] introduces a hash-based algorithm for parallel writing of non-zero gradients to memory, it does not address the imbalanced communications in distributed training and cannot handle hash collisions, resulting in information loss. In contrast, ZEN achieves balanced communications and avoids information loss.

7 Conclusion

We make two primary contributions in this work. First, we systematically explore the design space to identify optimal synchronization schemes for sparse tensors in distributed training. Second, we propose ZEN, a practical design to pursue optimal synchronization schemes with a novel hashing algorithm using parallel computing on GPUs without information loss. ZEN greatly improves the efficiency of sparse DNN model training without any impact on model accuracy.

Acknowledgment

We thank the anonymous reviewers for providing valuable and insightful feedback. This work is partially supported by the NSF under CNS-2214272.

References

[1] Billion Word Benchmark. https://code.google.com/archive/p/1-billion-word-language-modeling-benchmark/.

[2] Criteo dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[3] IWSLT 2014 De-En dataset. https://sites.google.com/site/iwsltevaluation2014/data-provided.

[4] NVIDIA NCCL. https://developer.nvidia.com/NCCL, 2021.

[5] Microsoft MSCCL. https://github.com/microsoft/msccl, 2023.

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

[7] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. 2017.

[8] Mohammadreza Alimohammadi, Ilia Markov, Elias Frantar, and Dan Alistarh. L-GreCo: Layerwise-adaptive gradient compression for efficient and accurate deep learning. *arXiv preprint arXiv:2210.17357*, 2022.

[9] Austin Appleby. Murmurhash 2.0, 2008.

[10] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Optimized broadcast for deep learning workloads on dense-GPU InfiniBand clusters: MPI or NCCL? In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–9, 2018.

[11] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. In *Proceedings of the twenty-sixth annual ACM symposium on theory of computing*, pages 593–602, 1994.

[12] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel DNN training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 359–375, 2021.

[13] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*, 35(6):1350–1385, 2006.

[14] Zhiruo Cao, Zheng Wang, and Ellen Zegura. Performance of hashing-based schemes for internet load balancing. In *Proceedings of IEEE INFOCOM 2000.*, volume 1, pages 332–341. IEEE, 2000.

[15] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, 1977.

[16] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. MONGOOSE: A learnable LSH framework for efficient neural network training. In *International Conference on Learning Representations (ICLR)*, 2021.

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[18] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[19] Artur Czumaj, Chris Riley, and Christian Scheideler. Perfectly balanced allocation. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 240–251. Springer, 2003.

[20] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, Eva Rotenberg, and Mikkel Thorup. The power of two choices with simple tabulation. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1631–1642. SIAM, 2016.

[21] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2021.

[22] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[23] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 676–691, 2021.

[24] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

[25] William Gropp. MPICH2: A new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 7–7. Springer, 2002.

[26] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: a factorization-machine based neural network for CTR prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1725–1731, 2017.

[27] Wenchen Han, Shay Vargaftik, Michael Mitzenmacher, Brad Karp, and Ran Ben Basat. Beyond throughput and compression ratios: Towards high end-to-end utility of gradient compression. *arXiv preprint arXiv:2407.01378*, 2024.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

[29] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 463–479, 2020.

[30] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[31] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[32] Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In *Advances in neural information processing systems*, pages 3294–3302, 2015.

[33] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 741–761, 2021.

[34] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. PyTorch Distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12).

[36] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-AI: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 766–775, 2023.

[37] Shengwei Li, Zhiquan Lai, Dongsheng Li, Yiming Zhang, Xiangyu Ye, and Yabo Duan. EmbRace: Accelerating sparse communication for distributed training of deep neural networks. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[38] Shigang Li and Torsten Hoefler. Near-optimal sparse allreduce for distributed deep learning. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–149, 2022.

[39] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *The International Conference on Learning Representations (ICLR)*, 2017.

[40] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, 2015.

[41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[42] Tharun Medini, Beidi Chen, and Anshumali Shrivastava. Solar: Sparse orthogonal learned and random embeddings. In *International Conference on Learning Representations*, 2020.

[43] Mellanox. Mellanox Corporate Update. https://www.mellanox.com/related-docs/company/MLNX_Corporate_Deck.pdf, 2022.

[44] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182*, 2017.

[45] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[46] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[47] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[48] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[49] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[50] NVIDIA. A Timeline of Innovation for NVIDIA. ttps://www.nvidia.com/en-us/about-nvidia/corporate-timeline/, 2021.

[51] OpenAI. AI and Compute. https://openai.com/blog/ai-andcompute/, 2021.

[52] Heng Pan, Penglai Cui, Zhenyu Li, Ru Jia, Penghao Zhang, Leilei Zhang, Ye Yang, Jiahao Wu, Jianbo Dong, Zheng Cao, Qiang Li, Hongqiang Harry Liu, Mathy Laurent, and Gaogang Xie. Enabling fast and flexible distributed deep learning with programmable switches. *arXiv preprint arXiv:2205.05243*, 2022.

[53] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[54] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[55] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[56] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.

[57] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, 2021.

[58] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[59] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[60] Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772*, 2019.

[61] Shaohuai Shi, Qiang Wang, Kaiyong Zhao, Zhenheng Tang, Yuxin Wang, Xiang Huang, and Xiaowen Chu. A distributed synchronous sgd algorithm with global top-k sparsification for low bandwidth networks. In *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2238–2247. IEEE, 2019.

[62] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[63] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*, 2018.

[64] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[65] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[66] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1), 2005.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[68] MK Vijaymeena and K Kavitha. A survey on similarity measures in text mining. *Machine Learning and Applications: An International Journal*, 3(2):19–28, 2016.

[69] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 149–164, 2023.

[70] Zheng Wang, Yuke Wang, Boyuan Feng, Guyue Huang, Dheevatsa Mudigere, Bharath Muthiah, Ang Li, and Yufei Ding. OPER: Optimality-guided embedding table parallelization for large-scale recommendation model. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 667–682, 2024.

[71] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[72] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. Hi-speed DNN training with Espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 867–882, 2023.

[73] Zhuang Wang, Xinyu Wu, Zhaozhuo Xu, and T. S. Eugene Ng. Cupcake: A compression scheduler for scalable communication-efficient distributed training. *Proceedings of Machine Learning and Systems*, 5, 2023.

[74] Zhuang Wang, Zhaozhuo Xu, Xinyu Wu, Anshumali Shrivastava, and T. S. Eugene Ng. DRAGONN: Distributed randomized approximate gradients of neural networks. In *International Conference on Machine Learning*, pages 23274–23291. PMLR, 2022.

[75] Maurice Weber, Daniel Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, et al. RedPajama: an open dataset for training large language models. *arXiv preprint arXiv:2411.12372*, 2024.

[76] Udi Wieder et al. Hashing, load balancing and multiple choice. *Foundations and Trends® in Theoretical Computer Science*, 12(3–4):275–379, 2017.

[77] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. GRACE: A compressed communication framework for distributed machine learning. In *Proc. of 41st IEEE Int. Conf. Distributed Computing Systems (ICDCS)*, 2021.

[78] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Toward more efficient training of deep networks. In *International Conference on Learning Representations, 2019*, 2020.

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

[80] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. MiCS: Near-linear scaling for training gigantic model on public. *Proceedings of the VLDB Endowment*, 16(1):37–50, 2022.

[81] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023.

Appendix

A Algorithm for a Strawman Solution

In this section, we present Algorithm 3, which is the algorithm for the data-independent solution with a straightforward hashing algorithm. This algorithm supplements the discussions in Section 3.2.

Algorithm 3: A strawman solution with hashing

Input: G is a dense tensor and $I \subset \mathbb{N}_+$ is a set of indices of its non-zero gradients. $n \in \mathbb{N}_+$ is the number of partitions. $r \in \mathbb{N}_+$ is the memory size for each partition. $h : \mathbb{N}_+ \to [nr]$ is an universal hash function.
Output: The partitioned sparse tensors.

```
Function Main(I, G, h(·)):
    Allocate memory x ← 0^{n×r}
    foreach idx ∈ I in parallel do
        p ← ⌊h(idx)/r⌋
        q ← h(idx) mod r
        x[p][q] ← idx
    end
    output = []
    for i ← 0 to n − 1 do
        indices = nonzero(x[i])
        values = G[indices]
        output.append((indices, values))
    end
    return output;
```

A naive solution to address Problem 1 is to apply a universal hash function across multiple threads on a GPU, i.e., each thread independently operates hash functions on a disjoint input and writes them into a hash memory. Given a dense tensor G, the set of indices of its non-zero gradients is I. The algorithm first allocates a memory x with shape $n \times r$, where n represents the number of partitions and r is the memory size for each partition. For every $idx \in I$, it uses a given universal hash function [15] $h : \mathbb{N}_+ \to [nr]$ to generate the hash value $h(idx)$, where nr is the range of hash function h. Next, it writes idx to the $(h(idx) \mod r)$th location in partition $\lfloor h(idx)/r \rfloor$. The hashing operation is performed in parallel to minimize the computation overhead [74]. After that, it extracts the non-zero indices from the memory of each partition and uses them to look up the corresponding gradients from G. Finally, it returns a sparse tensor for each partition and pushes them to the corresponding servers.

B Theoretical Analysis

B.1 Proof of Theorem 1

Theorem 1 (Optimal schemes)**.** *To minimize communication time for sparse tensors, the optimal synchronization scheme is either Balanced Parallelism or Hierarchical Centralization.*

We prove Theorem 1 with two lemmas.

Lemma 1. *When the partition pattern is fixed to Parallelism, the optimal scheme is Balanced Parallelism.*

Proof. There are three communication patterns: Ring, Hierarchy, and Point-to-point. We first consider synchronization schemes with [Point-to-point communication and Parallelism], namely, the PS architecture. Given n servers and a gradient tensor G with the density of d_G. We first analyze the communication time of Push and Pull operations separately. We then discuss the communication time of different PS schemes.

Push. Because the skewness ratio is s_G^n, the largest density in the n partitions is $s_G^n d_G$. The size of the sparse tensor extracted from this partition is $2s_G^n d_G M/n$. As a result, the communication time of Push in sparse PS is $2(n-1)s_G^n d_G M/n/B$.

Pull. After aggregation, the largest density in the n partitions becomes $s_G^n d_G^n$. In existing implementations of the PS architecture, the communication time of Pull is $2(n-1)s_G^n d_G^n M/n/B$ because each server needs to broadcast its aggregated results to all the workers with Point-to-point communications [29, 34]. In theory, there are other ways to implement Pull in the PS architecture. For example, each server can perform a `broadcast` collective operation. The performance of `broadcast` with different algorithms is analyzed in [10] and its communication time for Pull can be expressed as $2bd_G^n M/B$, where b is the number of rounds in an algorithm. For example, $b = \lceil \log n \rceil$ when it uses Binomial Tree Algorithm and $b = \frac{2(n-1)}{n}$ when it uses Scatter-AllGather Algorithm [10, 25].

Sparse PS. Combining the communication time of Push and Pull with Point-to-point communications, it overall communication time is $2(n-1)(d_G + d_G^n)s_G^n M/n/B$.

Sparse PS with broadcast. When considering `broadcast` for Pull, the overall communication time becomes $2(n-1)s_G^n d_G M/n/B + 2bd_G^n M/B$. We denote this case as Sparse PS with broadcast.

Balanced Parallelism. In Balanced Parallelism, the skewness ratio s_G^n is always 1. We replace the s_G^n in the communication time of sparse PS as 1 and have the communication time for Balanced Parallelism: $2(n-1)(d_G + d_G^n)M/n/B$.

Balanced Parallelism is optimal among schemes with Parallelism. It is clear that Balanced Parallelism is much better than sparse PS when the skewness ratio is large. The performance ratio of Spase PS with broadcast to Balanced Parallelism is $\frac{s_G^n}{1+\gamma_G^n} + \frac{n}{n-1}\frac{b\gamma_G^n}{1+\gamma_G^n} > \frac{s_G^n + b\gamma_G^n}{1+\gamma_G^n}$. Because both s_G^n and b are greater than 1, the ratio is also greater than 1. Hence, Balanced Parallelism always outperforms Sparse PS and Sparse PS with broadcast in terms of communication time.

We then prove that Balanced Parallelism outperforms other schemes. Because One-shot aggregation cannot leverage the overlaps among sparse tensors, the performance of synchronization schemes with One-shot aggregation is worse than those with Incremental aggregation. Therefore, we only consider Incremental aggregation for schemes with Ring commu-

nication or Hierarchy communication. We consider the best case for them, i.e., the skewness ratio is 1 after tensor partition with Parallelism. In addition, we only need to compare the first step because they have the same communication time in the second step. The communication time of the first step in Balanced Parallelism is $2(n-1)d_GM/n/B$.

Schemes with Ring and Incremental Aggregation. They have $n-1$ communication stages. The tensor density in the i_{th} stage is d_G^i. Note that $d_G^1 = d_G$. Therefore, the communication time is $2\sum_{i=1}^{n-1} d_G^i M/n/B$. Because tensors can get denser after aggregation, we have $d_G^i \leq d_G^j$ when $i < j$ and $\sum_{i=1}^{n-1} d_G^i \geq (n-1)d_G$. As a result, the communication time of schemes with Ring and Incremental aggregation is no less than that of Balanced Parallelism.

Schemes with Hierarchy and Incremental aggregation. They have $\log n$ communication stages. Because each partition has a hierarchical structure, the traffic volume in the i_{th} stage is $\frac{d_G^{2^{i-1}}}{2^{i-1}}Mn$ and the total traffic volume in all $\log n$ stages is $V = \sum_{i=1}^{\log n} \frac{d_G^{2^{i-1}}}{2^{i-1}}Mn$. Because $d_G^{2^{i-1}} \geq d_G$, $V \geq \sum_{i=1}^{\log n} \frac{d_G}{2^{i-1}}Mn = 2(n-1)d_GM$. Therefore, the traffic volume received at each GPU is no less than $2(n-1)d_GM/n$ and they are not better than Balanced Parallelism. □

Lemma 2. *When the partition pattern is fixed to Centralization, the optimal scheme is Hierarchical Centralization.*

Proof. When any two sparse tensors have no overlaps, the minimum traffic volume each GPU has to receive is all the tensors from other GPUs. Any synchronization scheme with Centralization achieves the optimal communication time.

When sparse tensors overlap, let n denote the number of GPUs and $I_0, I_1, \ldots, I_{n-1}$ the set of indices for non-zero gradients in each GPU, respectively. C is the overlap of all the sparse tensors, i.e., $C = \bigcap I_i$. If a synchronization scheme adopts Point-to-point communication or One-shot aggregation, each GPU has to receive C for $n-1$ times. Then we consider Incremental aggregation and the communication pattern is Ring or Hierarchy. With Ring, the tensor from each GPU is aggregated at each stage and then forwarded to the next GPU. Consequently, this tensor is received by every GPU. Because C is the common overlap, each GPU also has to receive C for $n-1$ times. When the communication pattern is Hierarchy, each GPU receives data from all the other GPUs with its own hierarchical structure that has $\log n + 1$ levels, as shown in Figure 4b. Because the root GPU is in each level, it has to receive C for $\log n$ times. It suggests that the traffic volume in the scheme with [Hierarchy, Incremental aggregation, and Centralization] is less than that in other schemes with Centralization. Let C' denote the overlap of a subset of the sparse tensors, we can draw a similar conclusion that a subset of GPUs has to receive C' multiple times. In other words, each GPU still has to receive the overlaps multiple times. □

Lemma 1 and Lemma 2 imply Theorem 1.

B.2 Proof of Theorem 2

Theorem 2 (Load Balance of Algorithm 1)**.** *Given a dense tensor G with $|G|$ parameters. Algorithm 1 provides a mapping $f : I \to [n]$ such that*

1. *With probability at least $1 - 1/n$, its imbalance ratio of Push is at most $1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G}})$.*
2. *With probability at least $1 - 1/n$, the imbalance ratio of Pull is at most $1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G^n}})$.*

Proof. The imbalance ratio of Algorithm 1 is only determined by $h_0 : \mathbb{N}_+ \to [n]$, while the set of hash functions H focuses on exact memory write.

The number of indices in I_i is $|G|d_G$, where d_G is the density of G. Since h_0 is data independent, part 1 in Problem 1 can be formulated as: given $|G|d_G$ balls, we would like to toss them into n bins with the universal hash function h_0. Taking the results from [11], the maximum load of the bins is at most $\frac{|G|d_G}{n} + \Theta(\sqrt{|G|d_G \log n/n})$ with probability at least $1 - 1/n$. Recall the definition of the imbalance ratio of Push in Definition 8:

$$Push_{h_0}^n = \max_{i,j \in [n]} \frac{n|I_i^j|}{|I_i|}.$$

Because $\max\{|I_i^j|\} \leq \frac{|G|d_G}{n} + \Theta(\sqrt{|G|d_G \log n/n})$, we have

$$\begin{aligned} Push_{h_0}^n &\leq \frac{|G|d_G + \Theta(\sqrt{|G|d_G n \log n})}{|G|d_G} \\ &= 1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G}}), \end{aligned}$$

with probability at least $1 - o(1)$. Thus, we finish the proof of the first part.

Since h_0 is data independent, part 2 in Problem 1 can be formulated as: given $|I| = |G|d_G^n$ balls, we would like to toss them into n bins with the universal hash function h_0. The maximum load on the bins is at most $\frac{|G|d_G^n}{n} + \Theta(\sqrt{|G|d_G^n \log n/n})$ with probability at least $1 - 1/n$. Recall the definition of the imbalance ratio of Pull in Definition 8:

$$Pull_{h_0}^n = \max_{i \in [n]} \frac{n|I_i'|}{|I|}.$$

Because $\max\{|I_i'|\} \leq \frac{|G|d_G^n}{n} + \Theta(\sqrt{|G|d_G^n \log n/n})$, we have

$$\begin{aligned} Pull_{h_0}^n &\leq \frac{|G|d_G^n + \Theta(\sqrt{|G|d_G^n n \log n})}{|G|d_G^n} \\ &= 1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G^n}}), \end{aligned}$$

with probability at least $1 - 1/n$. Thus, we finish the proof of the second part.

□



Extending Applications Safely and Efficiently

Yusheng Zheng
UC Santa Cruz

Tong Yu
eunomia-bpf Community

Yiwei Yang
UC Santa Cruz

Yanpeng Hu
ShanghaiTech University

Xiaozheng Lai
South China University of Technology

Dan Williams
Virginia Tech

Andi Quinn
UC Santa Cruz

Abstract

This paper presents the Extension Interface Model (EIM) and bpftime, which together enable safer and more efficient extension of userspace applications than the current state-of-the-art. EIM is a new model that treats each required feature of an extension as a resource, including concrete hardware resources (e.g., memory) and abstract ones (e.g., the ability to invoke a function from the extended application). An extension manager, i.e., the person who manages a deployment, uses EIM to specify only the resources an extension needs to perform its task. bpftime is a new extension framework that enforces an EIM specification. Compared to prior systems, bpftime is efficient because it uses extended Berkeley Packet Filter (eBPF)-style verification, hardware-supported isolation features (e.g., Intel MPK), and dynamic binary rewriting. Moreover, bpftime is easy to adopt into existing workflows since it is compatible with the current eBPF ecosystem. We demonstrate the usefulness of EIM and bpftime across 6 use cases that improve security, monitor and enhance performance, and explore configuration trade-offs.

1 Introduction

Developers extend their software to customize it for the needs of a particular deployment. For example, extensions can improve application performance [2, 40], add custom features [48, 50], enhance security [46, 62], and enable application observability for performance monitoring [61, 77] and debugging [39, 45]. Software extensions do not require modifying the original application, which enables customization while ensuring that the deployment can easily integrate maintenance updates from upstream repositories. Many software applications support extensibility, including web browsers [21, 43], HTTP servers [17, 25], text editors [24, 64], and databases [49, 59].

To extend software, a developer defines new logic for the program as a set of extensions, and associates each extension with a specific location in the host application, called an *extension entry*. When a user invokes the application, the system loads the host application and the user's configured extensions. Each time an application thread reaches an extension entry, the thread jumps to the associated extension. It executes the extension in the extension runtime context; once the extension completes, the thread returns to the host at the point immediately after the extension entry.

This paper presents a new approach for specifying the interface between an extension and a host, called the *Extension Interface Model* (EIM), and a new extension runtime system, called bpftime, that together provide safer and more efficient software extensions. EIM and bpftime are motivated by the challenges current extension frameworks face in balancing key extensibility features.

First, current extensibility frameworks struggle to navigate the tradeoff between extension interconnectedness and safety. On the one hand, enabling extensions that perform meaningful work requires *interconnectedness*, i.e., the ability to observe or modify the host application's state and to execute functions defined by the host. On the other hand, application deployments require that their extensions are safe, i.e., that an extension failure cannot harm the health of the host application and the underlying system. Achieving safety requires restricting an extension's behavior to limit both the system resources it consumes (e.g., the memory it uses, the files it opens) and the host interactions it performs (e.g., the host state it reads). Interconnectedness and safety are in tension because interconnectedness often necessitates allowing an extension to modify the host in potentially unsafe ways. We introduce the term *extension manager* to describe the person who is responsible for configuring the extensions on a given deployment and thus must navigate the tradeoff between interconnectedness and safety.

To maximize system safety, an extension manager should follow the principle of least privilege, granting extensions only the minimal set of features their use cases require. For example, a deployment that supports observability extensions for debugging and monitoring may have different interconnectedness/safety needs than one that supports extensions for customizing application behavior. Unfortunately, current

extension frameworks poorly support specifying and enforcing such deployment-specific interconnectedness/safety tradeoffs. Many frameworks (e.g., safe language runtimes [23, 38], NaCl [75]) cannot express an interconnectedness/safety tradeoff. Instead, they rely on applications to enforce their own safety, which is ad hoc and error-prone (see §2). Other frameworks (e.g., lwC [37], RLBox [44], Shreds [11]) do not support fine-grained limits on extensions. They either cannot restrict certain unsafe extension behaviors or cannot do so on a per-extension-entry basis. Finally, some frameworks (e.g., Orbit [31], Wedge [7]) do support fine-grained interconnectedness/safety tradeoffs, but they are not designed for extensibility and require modifying host application source code to impose different tradeoffs.

Our first contribution, EIM, supports fine-grained interconnectedness/safety tradeoffs. EIM's key idea is to represent the extension features needed for interconnectedness or restricted for safety through a single abstraction called a *resource*. For example, a resource can represent an extension's ability to call a host function or read a host variable. EIM represents the ability to use resources with capabilities [55, 60, 71]. An EIM specification is produced by two parties: the original application developer and the extension manager. First, developers define capabilities that represent the resources the host application can provide to extensions, essentially enumerating the extension interconnectedness that the host application supports. Then, during deployment, the extension manager creates extension classes that specify the set of capabilities allowed at a particular extension entry, essentially choosing the interconnectedness/safety tradeoff for each extension entry. In sum, EIM specializes interfaces for fine-grained protection, such as those for access control of OS objects (e.g., SELinux [54]) or browser manifest files [22], to support the fine-grained tradeoffs necessary for software extensions.

EIM specifications are runtime-agnostic, and we could enhance an existing extension framework to enforce them. However, current extension frameworks poorly navigate the tradeoff between three properties: extension safety, as specified in an interconnectedness/safety tradeoff; extension isolation, which prevents a host application from harming an extension and is necessary for security monitoring extensions; and extension efficiency, which requires that extensions execute at near-native speed. Current frameworks are inefficient because they employ heavyweight techniques for isolation and safety. For example, many frameworks (e.g., Orbit [31], lwC [37], Wedge [7]) provide new operating system-level isolation abstractions and require context-switch-like overhead to switch between a host and an extension. Other frameworks (e.g., Wasm [23], NaCl [75]) enforce safety and isolation using software fault isolation (SFI), which is much slower than native execution [29].

Our second contribution is bpftime, a new extension runtime that efficiently supports EIM and extension isolation using two design principles. First, the system uses lightweight approaches to provide extension safety and isolation. It enforces the safety in an EIM specification without any runtime overhead using extended Berkeley Packet Filter (eBPF)-style verification, and it enforces isolation with minimal overhead using ERIM-style intraprocess hardware-supported isolation [66, 72]. Second, bpftime introduces concealed extension entries, which improve efficiency by eliminating runtime overhead from extension entries that are not in use by a running process. Concealed extension entries use binary rewriting [10, 14, 18, 70] to inject an extension entry into a host only when a user loads an associated extension. While prior work uses techniques similar to bpftime's verification, isolation, and rewriting techniques, bpftime is the first system combine them to satisfy EIM's requirements.

Additionally, bpftime is fully compatible with eBPF, streamlining the system's path to adoption. With eBPF compatibility, not only can current users of eBPF extensions (e.g., uprobes) seamlessly adopt bpftime[1], but bpftime extensions can also share state and interact closely with eBPF kernel extensions, thereby supporting extensibility use cases that require extending both the kernel and applications.

We maintain bpftime as an open source project [2]; bpftime has 1,000 stars on GitHub, more than 20 contributors, and several PRs per month. Our users currently rely on bpftime and EIM for many use cases, including observability, fault injection, hot patching, and other application customizations. Inspired by these users, we present 6 use cases that highlight the benefits of bpftime and EIM. These use cases explore design tradeoffs, improve security, monitor performance, and enhance system efficiency. In particular, we use bpftime to monitor a microservice application, create new durability configurations in Redis, cache metadata operations in FUSE, implement an SSL-supporting distributed tracing tool, monitor system calls for performance analysis, and enhance webserver security.

We evaluate bpftime's performance on the aforementioned 6 use cases. We find that bpftime improves the throughput of profiling microservices by a factor of 1.5 compared to eBPF. bpftime enables a Redis durability configuration that loses orders of magnitude less data in a crash while decreasing throughput by only about 10%. bpftime enables FUSE caching that accelerates operations by orders of magnitude. The system adds only 2% overhead when extending Nginx, which is up to 5× lower than state-of-the-art alternatives such as WebAssembly, Lua, ERIM [66], and RLBox [44]. bpftime reduces the overhead of SSL traffic monitoring by a factor of 3.79 compared to native eBPF. Moreover, bpftime offers configurations that prevent monitoring overhead from affecting unmonitored processes, a feature eBPF cannot provide. We also use microbenchmarks to illustrate the key features that enable bpftime's performance advantages and demonstrate bpftime's compatibility with eBPF.

[1] Note: eBPF uprobes do not support all bpftime features.

[2] available at https://github.com/eunomia-bpf/bpftime.

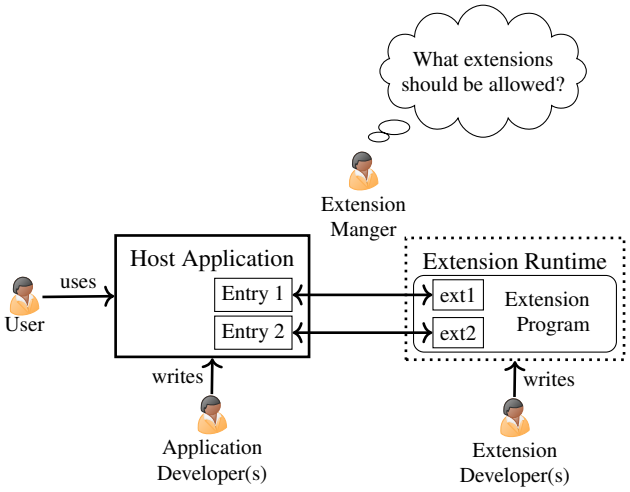


Figure 1: A process extended with two extensions, ext1 and ext2, with associated extension entries, entry1 and entry2, respectively. The application developer(s) and extension developer(s) write the host application and extension program, respectively. The User uses the host application. The Extension Manager decides which extensions to allow and use in the deployment.

In sum, our contributions are:

- EIM, which allows users to specify fine-grained interconnectedness/safety tradeoffs.
- bpftime, an efficient extension runtime system that enforces EIM specifications and isolation through two separate verification techniques and concealed extension entries.
- An evaluation of EIM and bpftime demonstrating their usefulness and efficiency in 6 use cases.

In the rest of the paper, we motivate EIM and bpftime (§2), describe EIM (§3), explain the design and implementation of bpftime (§4), discuss 6 use-cases (§5), evaluate bpftime(§6), describe related work (§7), and conclude (§8).

2 Motivation

System extensions augment an application without modifying its source code to customize behavior, enhance security, add custom features, and observe behavior. By supporting application modifications without requiring source code changes, extensions allow a customized deployment to integrate maintenance updates from upstream repositories easily and can provide assurances of security and safety. The rest of this section discusses the principal roles of system extensions (2.1), provides an example web-server use-case (2.2), articulates the key properties of extension frameworks (2.3), discusses limitations of the current state-of-the-art (2.4), and articulates the threat model (2.5).

2.1 Roles

The system extension usage model considers four key principals. The *application developers* are a group of trusted developers who write the original application, while the *extension developers* are a group of trusted developers who create the extensions. System extensions assume that both the application developer(s) and extension developer(s) are trusted but fallible, so applications and extensions might be exploitable but are not intentionally malicious. Next, the system extension model includes an *extension manager*, a trusted individual that installs and manages the extensions; the model relies on the manager to be both trusted and infallible. Finally, *users* are untrusted individuals who interact with the extended application; users can be malicious and may try to craft inputs that would trigger vulnerabilities in otherwise benign code.

Figure 1 provides a representation of an extended application and shows the role of each principal. The application developers write the host application. The extension developer creates the extension program, which can read and write application state and execute application-defined functions. The extension manager is responsible for deciding which extensions to use at each extension entry. Finally, users produce input that interacts with the host application and, indirectly, the extension program.

2.2 Web-Server Example

Consider an instance of Nginx deployed as a reverse proxy. The application developers write the server, while the extension developers provide a suite of possible extensions to deploy on the system for monitoring, firewalls, and load balancing. The extension manager determines the extensions for the deployment and the privileges to provide each extension. First, the manager uses an extension program that monitors traffic to detect reliability issues [27]. Second, the manager deploys an extension program that implements a firewall that returns a 404 response for URLs that are indicative of SQL injection and cross-site scripting attack. Finally, the manager deploys an extension program to perform load balancing across the possible servers downstream from the proxy by periodically contacting downstream servers to measure system load [26].

2.3 Key Extension Framework Features

Extension use-cases require three key features:

Fine-grained Safety/Interconnectedness tradeoffs. Extensions must be *interconnected*, i.e., able to interact with the host application. Host interactions include reading/writing host state and executing host-defined functions. The Nginx example extension programs highlights both of these features: extensions in the observability use-case require read access to host state, while the firewall, load balancing, and statistics

Bug	Software	Summary
Bilibili [73]	Nginx	Livelock (infinite loop) in an extension caused production outage.
CVE-2021-44790 [47]	Apache	Buffer overflow in httpd's lua module causes application to crash.
CVE-2024-31449 [42]	Redis	Stack overflow in Lua script leads to arbitrary remote code execution.

Table 1: Example issues caused by extension safety violations.

extensions call Nginx functions to parse user requests and return responses.

At the same time, an extension manager wishes to ensure that extensions are *safe*, i.e., do not harm the health of the deployment. Safety does not aim to thwart malicious extensions, but rather to ensure that a bug in the extension cannot harm the reliability or security of the host. Extension bugs have lead to catastrophic consequences in production, including livelock, system crashes, and remote code execution; Table 1 provides examples.

There is no single definition of safety appropriate for all extension use-cases, since safety and interconnectedness are in tension. Instead, the manager should be able to follow the principle of least privilege and allow each extension to perform only the actions necessary for the interconnectedness required for its task. For example, the Nginx observability extensions only need to read from specific host states, but would be unsafe if allowed to write to them. In contrast, the Nginx firewall extension needs to read/write to a different set of host states, but would be unsafe if it were allowed to read the observability states.

Safety rules that can be applied to each individual extension are *fine-grained safety/interconnectedness tradeoffs*. We note two desirable properties. First, the same extension entry may need to support different safety/interconnectedness tradeoffs since it may be useful for separate extension use-cases. Second, while the universe of possible interconnectedness features depends on the host application and thus requires a knowledgeable application developer to modify the host source code, the manager should be able to modify the safety/interconnectedness tradeoffs for their extensions without changing the original application.

Isolation. Extensions must be *isolated*, i.e., not be harmed by the host application. Isolation does not thwart a malicious host application, but rather ensures that attackers cannot circumvent extension-based security by exploiting bugs in the host application. Isolation requires ensuring that host applications cannot modify extension states. The Nginx firewall extension is an example that relies upon isolation.

Efficiency. Extensions should be *efficient*, i.e., execute at near-native speed, since they may be deployed on the hot path of production systems. For example, the Nginx load balancer extension is invoked on the hot path for all user requests.

2.4 The Limitations of State-of-the-Art

We describe the limitations of existing extension frameworks.

Native Execution. A number of extensibility approaches execute the application and extension in the same execution context, essentially treating the extension as a component of the original program. Such approaches include those that use dynamic loading (e.g., `LD_PRELOAD`) and dynamic binary instrumentation [8, 39, 45]). These systems provide efficiency but neither isolation nor support fine-grained safety/interconnectedness tradeoffs.

SFI-based tools. Many extension frameworks use SFI [69] to provide isolation, including XFI [16], NaCL [75], RLBox [44], and language-based sandboxes such as WebAssembly [23] and Lua [38]. Some of these tools (e.g., Lua, WebAssembly, and NaCL) do not provide an interface for safety/interconnectedness tradeoffs. Instead, they rely on the host application to check for safety violations, an approach that has proven buggy (see Table 1). Other tools (e.g., RLBox, XFI) lack fine-granularity or the ability to limit certain extension behaviors. Additionally, SFI-based tools are typically inefficient since they validate extension behavior at runtime [29].

Subprocess Isolation. Subprocess isolation systems, such as Wedge [7], Shreds [11], lwC [37], and Orbit [31], separate extensions from the host application through operating system isolation abstractions. Such systems ensure isolation. However, some lack fine-grained interconnectedness/safety tradeoffs (Lwc and Shreds), while others (Orbit and Wedge) could provide such tradeoffs, but only after code changes to the host application, because they are not designed for extensibility. Finally, such systems struggle to be efficient for frequently-executed extension use-cases, since they require context-switch-like overheads when switching between the host application and its extensions.

eBPF uprobes. While it is usually used for kernel extensions, the extended Berkeley Packet Filter (eBPF) framework provides userspace extensions through the uprobe interface. eBPF uprobes are isolated from the host application, but do not support fine-grained interconnectedness/safety tradeoffs. Moreover, eBPF uprobes are not efficient. eBPF uprobes place a software breakpoint on every extension entry, causing the system to trap into the kernel to execute each extension.

Aspect-oriented programming.. Aspect-oriented programming allows extensions, but existing aspect-oriented languages do not support safety/interconnectedness tradeoffs. For example, if an AspectJ extension were exploited [34], the attacker would have unrestricted ability to observe and modify the original host application.

2.5 Threat Model

The system extension threat model is as follows. First, it assumes that the extension manager accurately and completely identifies the correct safety/interconnectedness tradeoff for each extension entry. This means that a buggy extension cannot corrupt, crash, or hang an application through the interface that the extension manager allows. Second, the model assumes that the control-flow of the application cannot be modified or corrupted even if an application is compromised—essentially equivalent to control flow integrity [1]—since an attacker could otherwise circumvent extension execution. Given these limitations, the threat model considers two key threats. First, it considers buggy extensions that accidentally crash or hang the application through errant pointers, infinite loops, or stack corruption that are outside of their allowed safety/interconnectedness tradeoff. Second, it considers compromised applications that modify or corrupt the state used by extensions to alter the extension's behavior.

3 Extension Interface Model (EIM)

The Extension Interface Model (EIM) is a new model for specifying fine-grained interconnectedness/safety tradeoffs. Using the model, an extension manager can follow the principle of least privilege and specify a tradeoff that would enable extensions to perform their tasks (i.e., sufficiently interconnected) with minimal potential harm to the system (i.e., are sufficiently safe). The model is sufficient to prevent past extension bugs from harming production; e.g., an extension manager can use EIM to prevent each of the bugs in Table 1. An extension framework (e.g., bpftime) can later ensure that the extensions loaded into an application follow their EIM specification, thereby ensuring their safety.

EIM's key idea is to represent the extension features that might be necessary for interconnectedness or restricted for safety as a *resource*. Such resources include both classical systems resources, such as compute cycles, and host application interactions, such as the ability to call a host function or read/write to a host variable. EIM models the ability to use a resource as a capability [55, 60, 71]. An EIM specification encodes fine-grained interconnectedness/safety tradeoffs by specifying the set of capabilities that a loaded extension is allowed to use when configured for a given extension entry.

An EIM specification consists of two separate components. First, the EIM specification includes a development-time configuration, prepared by an application developer, that specifies the set of possible safety and interconnectedness features (§3.1). Second, an EIM specification includes a deployment-time configuration, prepared by an extension manager, that specifies precise interconnectedness/safety tradeoffs (§3.2).

3.1 Development-time EIM Specification

```
1 State_Capability(
2 name = "readPid",
3 operation = read(ngx_pid))
4
5 Function_Capability(
6  name = "nginxTime",
7  prototype = (void) -> time_t,
8  constraints = {rtn > 0})
9
10 Extension_Entry(
11  name="processBegin"
12  extension_entry = "ngx_http_process_request",
13  prototype = (Request *r) -> int))
14 Extension_Entry(
15  name="updateResponseContent"
16  extension_entry = "ngx_http_content_phase",
17  prototype = (Request *r) -> int*)
```

Figure 2: An EIM development-time specification for a simplified version of the Nginx observability use-case.

The development-time EIM specification encodes the possible interconnectedness features and extension entries of the host application. Since these features are tightly coupled with the host application, the development-time EIM is created by an application developer while developing the host application. A development-time EIM specification defines three sets of entries: state capabilities, function capabilities, and extension entries. Figure 2 provides an example of a development-time EIM specification for the Nginx observability use-case.

State Capability. A state capability expresses the ability to read or write to a global state located in the host. A state capability includes a name and an operation of the form `read(var)` or `write(var)`, which specify the capability to dereference and read or write to a variable, `var`, respectively. Figure 2 includes a state capability, `readPid`, that enables reading the global variable in Nginx that stores the process id of the current worker process (Lines 1– 3).

Function Capabilities. A function capability expresses the ability to execute a host-provided function. An application developer defines a new function capability by providing the capability a name, the function prototype of the function, and a set of constraints that provide pre- and post-conditions for the function. Figure 2 includes a function capability in Lines 5–8. The capability specifies the ability to call Nginx's function for getting a timestamp. The function constraints provide the post-condition that its return value is positive.

Constraints. Constraints ensure that extensions safely use host-provided functions and can encode two things. First, constraints can encode binary relationships between arguments and return values, high-level semantic facts, and boolean operators over other constraints. Second, constraints can encode high-level fats about the function that an extension framework could support. Currently, EIM supports allocation facts indicating that a function's return was allocated, IO facts in-

```
1 Extension_Class(
2  name = "observeProcessBegin",
3  extension_entry = "processBegin",
4  allowed = {instructions<inf, nginxTime,
       readPid, read(r)})
5 Extension_Class(
6  name = "updateResponse",
7  extension_entry = "updateResponseContent"
8  allowed = {instructions<inf, read(r),
       write(r)})
```

Figure 3: A deployment-time EIM specification for the simplified Nginx observability use-case.

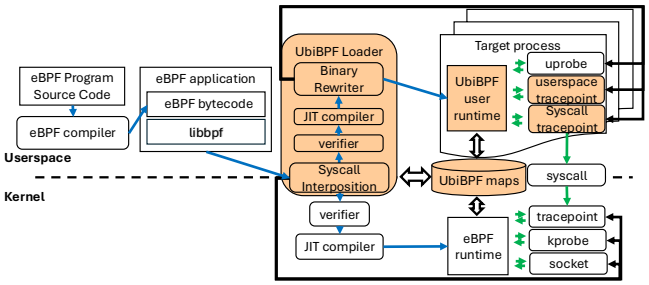


Figure 4: bpftime Design. White components are from eBPF; orange components are new to bpftime. Blue arrows show execution flow when compiling and loading an eBPF application. Green arrows show execution flow when an eBPF extension executes. White arrows with black outline indicate components that interact with eBPF maps

dicating that the function requires the capability to perform IO, annotation facts that indicate a relationship between arguments equivalent to those that linux provides through current eBPF annotations [35], and read/write facts indicating that the caller must hold read/write capabilities for a specified field within a function argument.

Extension Entry. An extension entry specifies the points in the host application that an extension can override. EIM supports extension entries that specify a function—i.e., extending an EIM extension entry replaces the definition of the associated function. An extension entry includes a name, the name of the original function from the host application, and the prototype of the function. Conveniently, many applications already include extensibility support through function interposition that an application developer can reuse for the EIM specification. However, developers may need to define and call new functions in the host application to support emerging extension use-cases. Figure 2 includes two extension entries, `processBegin`, for extending Nginx immediately after it is finished parsing a request, and `updateResponseConent`, for modifying the response that Nginx returns to a client.

3.2 Deployment-time EIM Specification

Deployment-time EIM specifications identify the interconnectedness/safety tradeoffs for a deployment. In particular, the specification identifies the capabilities that an extension requires for its task. Extension Managers write the deployment-time EIM specifications, since identifying the right tradeoff requires reasoning about a deployment's extension use-cases and available resources.

A deployment-time EIM specification includes a set of *Extension Classes*, each representing an interconnectedness/safety tradeoff at an extension entry. An extension class includes a name, the name of an extension entry from the development-time EIM specification, and a set of capabilities allowed by the class. An extension class' allowed capabilities can include capabilities outlined in the development-time EIM specification, state capabilities over the arguments in the extension entry's prototype, and resource capabilities, which specify an ability to consume a hardware resource. EIM provides two such resources, `instructions` and `memory`.

Figure 3 is a deployment-time EIM for the simplified Nginx observability use-case. The specification includes constraints that prevent extension bugs; e.g., it prevents the extension livelock that caused a production outage to an Nginx deployment [73]. The specification includes two extension classes, one for each of the extension entries defined in Figure 2. The allowed capabilities (Lines 4 and 8) include state capabilities (e.g., `read(r)`), resource capabilities (e.g., `instructions < inf`), and function capabilities (e.g., `nginxTime`).

4 bpftime Design

EIM specifications are platform independent, so we could expand existing extension frameworks to enforce them. However, existing frameworks provide safety and isolation through inefficient heavyweight operating-system supported isolation [7, 11, 31, 37] or software-fault isolation [23, 38, 44, 75]. Expanding such systems to enforce EIM's capability model would further degrade their efficiency and yield an unusual system.

So, we designed and implemented bpftime, a new extension framework for extending compiled applications (e.g., those written with C,C++,Go, etc.). bpftime efficiently enforces EIM specifications and supports extension isolation by employing two design constraints. First, the system uses separate lightweight approaches for EIM enforcement and for isolation, similar to how KFlex [15] uses two separate verification techniques for kernel extensions. Namely, bpftime enforces EIM safety/interconnectedness tradeoffs without imposing any runtime overhead by employing eBPF-style verification and provides efficient extension isolation using ERIM-style [66] intra-process hardware-supported isolation. Second, bpftime introduces concealed extension entries, which use binary rewriting [10, 14, 18, 70] to inject extensions into the

host application so that extension entries are zero-cost when not in use by an extension.

bpftime is compatible with eBPF: it provides nearly identical development model, execution models, and even reuses some architecture from eBPF. Thus, existing eBPF uprobe use-cases can seamlessly employ bpftime, and users can easy deploy bpftime extensions alongside kernel eBPF extensions. Alas, ensuring eBPF compatibility is challenging. Linux's eBPF ecosystem consists of tightly coupled components—compilers, runtime libraries, and the kernel—that are nearly impossible to disentangle. Instead of decoupling them, prior user-level eBPF systems [28, 53] re-implementing the entire eBPF technology stack and ultimately failed to provide reasonable performance and compatibility. Instead, bpftime identifies a narrow waist in the current eBPF ecosystem and interposes at this point. In particular, the system interposes on eBPF-related system calls and on the shared map mechanism for sharing data across extensions.

We implemented bpftime in 13,000 lines of code; Figure 4 provides the high-level design. bpftime's trusted computing base includes the kernel eBPF verifier, the binary rewriter, the operating system, and hardware-supported intra-process memory protections. Our current implementation only supports isolation on Intel x86, but different hardware vendors have similar technology that bpftime could employ to add support (e.g., ARM memory domains).

The rest of this section describes bpftime's usage model (§4.1), the bpftime loader (§4.2), and the bpftime runtime (§4.3).

4.1 bpftime Usage

bpftime follows a usage model similar to eBPF. Application developers first create development-time specifications for their application. Extension developers then write programs using eBPF. Extension managers later create deployment-time EIM, and load the extension into the host application with eBPF system calls. The bpftime loader intercepts these calls, validates extensions against their class-defined capabilities, initializes bpftime maps, sets up the bpftime runtimes, and attaches extensions to their extension entries. During execution, the runtime enforces extension isolation.

We elaborate on bpftime's support for EIM and bpftime extension programs.

4.1.1 bpftime support for EIM

To reduce the challenge of maintaining consistency between an application and its EIM specification, bpftime supports development-time EIM specifications through the use of annotations. Namely, application developers annotate their code using a set of annotations that are based upon eBPF's kfunc annotations [35] to specify state capabilities, function capabilities, and extension entries. During compilation, bpftime uses static analysis to extracts the development-time EIM specification from the annotations as well as symbol and type information from the application's debug symbols. The system encodes the resulting specification at the binary level that could, in principle, support any compiled language; the tool currently supports C/C++.

In addition to the application-specific development-time EIM specification, bpftime adds generic extension entries and function capabilities. The system adds `uprobe` and `uretprobe` execution entries for every host-defined function; the host logically executes these entries at the beginning and return of their affiliated function, respectively. Similarly, bpftime adds `sysenter` and `sysexit` execution entries for system calls. The prototype for the `uprobe` and `sysenter` entries include the arguments for the associated function or systemcall, respectively, while the prototype for `uretprobe` and `sysenter` includes the return value for the associated function or systemcall return, respectively. Additionally, bpftime defines a number of runtime-defined helper functions for common tasks and encodes them as function capabilities in the development-time EIM specification. For example, it adds a function capability to modify the return value for `uretprobe` and `sysexit` probes and separate capabilities to interact with bpftime-provided shared memory for process-local memory, inter-process memory, or process-kernel memory.

bpftime supports concealed extension entries when it generates the development-time EIM specification. The system removes the calls to all defined extension entries from the compiled program, both the developer-specified and automatically generated ones, thereby ensuring that any entries that are not used by an extension do not impose a runtime cost.

bpftime supports deployment-time EIM specifications through a YAML configuration language. In addition to extension classes that an extension manager defines, bpftime supports two built-in extension class types that auto-generate useful extension entries. First, bpftime supports observability, which adds an observability extension class for all auto-generated entries (i.e., `uprobe`s, `uretprobe`s, `sysenter`s, and `sysexit`s). Each observability extension class allows capabilities to read the entry's arguments, use a bounded number of instructions, and use a subset of the bpftime-provided function capabilities. Second, bpftime supports customizability, which adds customizability for all auto-generated entries. Each customizability extension class allows capabilities to read/write the entry's arguments using a bounded number of instructions and to use many of the bpftime-provided functions.

4.1.2 bpftime support for extension programs

bpftime supports extension programs that are similar to those supported by traditional eBPF. A developer writes an event-based program consisting of two parts: (1) a set of bpftime extensions and (2) a userspace control application that loads

and communicates with the extensions. Each user-defined extension specifies an associated extension class from EIM. Meanwhile, the eBPF application uses standard APIs (e.g., libbpf) to share data and interact with extensions through bpftime maps.

4.2 bpftime Loader

The bpftime loader prepares the extensions for execution by loading them into the host application and initializing the bpftime maps required for the eBPF application. To cope with the complexity of ensuring compatibility with eBPF, the loader is a thin interposition layer between eBPF applications and the kernel. The loader exposes the eBPF-related system calls interface from the eBPF ecosystem to userspace, but implements them using standard POSIX abstractions (files, unix sockets, etc.). Thus, bpftime supports control applications that use standard eBPF runtimes (e.g. bcc and libbpf) and are compiled using a standard eBPF compiler without requiring kernel modifications. The loader consists of two key components, the verifier, which implements bpftime's eBPF-style verification of extension safety, and the binary rewriter, which supports extensions that use concealed extension entries.

The Verifier. The verifier ensures that extensions are allowed by their associated extension class in the EIM specification. It receives each extension as an eBPF bytecode program. The verifier first performs a few basic verification checks: it ensures that an extension's function signature matches its extension class and that the extension only calls functions allowed by the extension class' function capabilities.

Next, the verifier converts the EIM specification into constraints that it adds to the eBPF bytecode program and uses the eBPF verifier to verify the extension. First, bpftime parses DWARF debugging information from the host application to produce BTF information to add types to the eBPF bytecode. The verifier supports function capabilities by replacing each call to a function capability in the extension with a mock new eBPF kfunc that it generates; using mock kfuncs allows bpftime to reuse the eBPF verifier's support for external functions. Additionally, the verifier converts all function capability constraints into assertion statements that it inserts into the eBPF bytecode before the generated kfunc. Next, the verifier uses the extension class' capability to add additional constraints. It supports state capabilities by modifying the function prototype (e.g., adding `const` for read only arguments) and encodes resource capabilities using verifier-supported clauses [3]. Finally, the verifier uses the verifier to ensures that the extension upholds the EIM specification, is memory and type safe, and will not execute a hardware exception.

Once verified, extensions pass through a userspace JIT compiler, which compiles them into native code. The JIT compiler passes the compiled extensions to the rewriter.

[3]The Linux kernel provides minimal support for custom resource constraints, so our implementation falls back to using PREVAIL [19] for these.

The Rewriter. The rewriter uses `ptrace` to pause the host application and load the bpftime user runtime (§4.3) into it, which allows the system to support use-cases that extend live applications; the system could use existing live-update solutions, such as MCR [20] and KUP [33], to improve performance. The rewriter instruments the program's instructions using Frida [18] and libcapstone [36] to ensure that extensions are invoked when the host reaches their associated extension entry. For extension entries that relate to a single instruction (e.g., `uprobes`, `uretprobes`), the rewriter uses a standard instruction trampoline: it replaces the instructions that were originally at the hook point with a `call` into a preamble for the point's associated eBPF function and places the overwritten instructions at the end of its preamble. Supporting extension entries that relate to system calls is more complex since an application can execute a system call instructions (e.g., `sysenter`) at any point throughout their application. Thus, the runtime iterates through all of the instructions in the application and places a trampoline on all system call instructions. Since `sysenter` is smaller than a trampoline, the system uses zpoline [74], which uses the zero page to accommodate two-byte call instructions. The current implementation only supports a single extension per entry point; users requiring multiple extensions can use a dispatcher pattern to individually call each extension, similar to the design used by libxdp.

4.3 bpftime Runtime

The bpftime runtime executes extensions in the same process as the original application to ensure efficiency. The runtime implements bpftime's intra-process isolation approach to provide extension isolation, which is less expensive than the isolation techniques of existing extension frameworks (e.g., software-fault isolation). Finally, the runtime implements bpftime maps, which are compatible with eBPF maps.

Intraprocess Isolation. The bpftime runtime uses ERIM-style intraprocess isolation to ensure extension isolation [66]. The runtime ensures that the application cannot modify extension logic by setting the memory pages that contain extensions and extension trampolines to be non-writable. The runtime ensures the integrity of extension memory using intra-process memory protections. During loading, the bpftime loader allocates a memory protection key for the extension's memory. The loader adds instructions that use the allocated key (i.e., `WRPKRU`) immediately before all userspace extensions, and adds instructions to reset the key immediately before returning to the original application. To protect the key itself, bpftime loads the key's value directly into the extension and sets the extension's memory permissions to be non-readable. The system is currently susceptible to the syscall-based attacks on ERIM [13], but could adopt Jenny's syscall filtering defenses [56] to resolve the issue.

bpftime Maps. Conventional eBPF provides maps to store

state across extension invocations. Unfortunately, eBPF maps support access to userspace programs through expensive system calls. So, bpftime provides a new map implementation to support system call free access across extension invocations, including bpftime extensions, eBPF kernel extensions, and bpftime control applications. Additionally, bpftime maintains compatibility with the existing eBPF ecosystem by interposing on eBPF map system calls and adding logic that uses bpftime maps instead of traditional eBPF ones. bpftime maps support three sharing modes: local process (nonshared), shared across multiple processes, and shared across processes and the kernel. bpftime maps provide a wide-range of data structures, including hash maps, arrays, LPM tries, ring buffers, perf event arrays, per-CPU variants. Additionally, bpftime maps are highly efficient: they offer per-cpu variants to remove contention and use lock-free synchronization when necessary.

5 Use Cases

We maintain bpftime as an open source project and have cultivated a small use-base. EIM and bpftime have many use-cases, including observability, hot patching, and security enforcement. Inspired in part by the experiences of our users, we craft 6 use-cases of EIM and bpftime that explore design tradeoffs, improve security, monitor performance, and improve performance.

Nginx Plugin. An extension manager manages an Nginx deployment that frequently receives suspicious traffic and wishes to deploy an extension to ensure that the system remains secure. While Nginx currently supports extensions through Lua and WebAssembly, the extensions framework offer poor safety/interconnectedness trade-off and impose high overhead since the extensions copy data to-and-for Nginx instead of operating over the same objects. Since Nginx is already designed for extensibility, creating the development time EIM specification only requires an application developer add annotations to current Nginx extension functions. The extension manager specifies the deployment time EIM by adding a single extension class to an extension entry that corresponds to processing request. The extension class supports reading from an argument representing the current request and writing to the return value. The manager then deploys an extension that implements a firewall to return a 404 response when a request's URL is indicative of SQL injection and cross-site scripting attacks.

sslsniff. Distributed tracing tools struggle to provide useful results when used on encrypted traffic. `sslsniff`, a tool from the bcc [52] project, allows an extension manager to intercepts all SSL/TLS data within userspace to enable better distributed tracing for encrypted data. `sslsniff` uses eBPF `uprobe`s on SSL/TLS encryption and decryption functions in OpenSSL. It suffers from high overhead due to eBPF `uprobe` inefficiencies. bpftime's automated `uprobe`s and observability extension classes enable `sslsniff` to work on bpftime without requiring changes.

Syscount. Syscount [3], another bcc tool, allows aggregating system call activity for a process on a running system by using system-wide extensions on `sysenter`s and `sysexit`s. To limit monitoring to a single process, the tool must observe the system calls of every process on the system and perform manual filtering in the extension. As a result, Syscount imposes overhead on every process, even those that are not monitored. An extension manager can run Syscount on bpftime without any modifications by using bpftime's automated `sysenter` extension entries and its automated observability extension classes. Since bpftime adds Syscount extensions into only the monitored process, the Syscount overhead is localized to the monitored processes.

DeepFlow. DeepFlow is an open-source observability platform for observing a microservice's behavior across the kernel and userspace with eBPF. Unfortunately, Deepflow can lead to as much as a 50% drop in application throughput due to the overheads of eBPF `uprobe`s. We ported DeepFlow to bpftime, modifying 10 out of the 5,000 lines of eBPF code, so that the tool could benefit from bpftime's efficiency. DeepFlow extensions use bpftime's automated uprobe and system call extension entries. DeepFlow uses the system's automated observability extension classes with one caveat: it adds support for using process-kernel shared maps.

FUSE Caching. The Filesystem in Userspace (FUSE) framework offers reliability and security advantages compared to in-kernel alternatives, but imposes considerable runtime overhead because it requires numerous additional context switches for every I/O systemcall [67]. ExtFuse [6] eliminates much of the overhead from using FUSE by enabling a FUSE filesystem to push its logic into the kernel. However, ExtFuse is invasive: it requires a custom kernel module that is difficult to maintain. In this use-case, an extension developer uses bpftime to implement and support the same benefits of ExtFuse without requiring a custom and difficult to maintain kernel model. They configure bpftime's automated customizability extension classes on the system's automated `syscall` tracepoints and add function capabilities for bpftime helpers that interact with file paths, such as `realpath`, and for using process-kernel shared maps. Then, they configure the deployment to use two extensions, written by an extension developer, that accelerate applications that use FUSE. The first is a metadata cache that accelerates repeated lookups to the same file system entries by extending `open`, `close`, `getdents`, and `stat`. It exploit's bpftime's compatibility with eBPF to maintain cache consistency using a kprobe extension on `unlink` in the kernel [4]. The second extension implements a blacklist to accelerate permission checking for functions that access filesystem entries (e.g., `open`).

Redis Durability Tuning. Redis, a key-value store, offers durability through a write-ahead log (called an Append-Only

[4]The second extension must operate in the kernel since non-extended processes may `unlink` a cached file.

```c
int writeback_fin_cnt;

SEC("user_define_ops/fsync_ext")
int BPF_UPROBE(start_fsync, int __fd) {
 if (writeback_fin_cnt == 0) {
   io_uring_wait_and_seen();
 } // else fsync is complete so no wait
 successful_writeback_count = 0;
 io_uring_prep_fsync(__fd);
 io_uring_submit();
 return 0;
}

SEC("kretprobe/file_write_and_wait_range")
int BPF_KPROBE(file_write_and_wait_range,
            struct file *file, loff_t start,
            loff_t end) {
 ... // Check if fsync was from redis/AOF
 __sync_fetch_and_add(&writeback_fin_cnt, 1);
 return 0;
}
```

Figure 5: The Redis Delayed fsync extension.

File (AOF)). By default, Redis offers three durability configurations that tradeoff durability and performance overhead: no AOF, which does not provide durability; everysec, which ensures that writes are durable every second; and alwayson, which ensures that every write is immediately durable. The durability gap between alwayson and everysec is substantial: a crash under everysec can lead to the loss of tens of thousands of updates. Unfortunately, alwayson also reduces throughput by a factor of six compared to everysec.

Redis can use bpftime to provide customizable durability through extensions. Since Redis AOF is not currently designed for extensibility, the application developer needs to add new functions to the source code to support extensibility. The developer defines three new functions and call them at the top of Redis's functions for `write`, `fsync`, and `fdatasync`. Then, they use bpftime annotations to identify the new functions as extension entries. Altogether, this change required about 20 lines of code.

With the updated Redis, the extension manager can explore a custom durability policy for their deployment. They first configure an extension that converts synchronous I/O operations into batched ones by using Linux's `io_uring`. The extension batches *b* I/O operations (calls to `write` and `fsync`) before waiting on them to complete, which ensures that the system loses at most *b* updates in an untimely crash. To use the extension, the extension manager creates two extension classes for `write` and `fsync` that have bounded computation and function capabilities to execute `io_uring` and bpftime map helpers for process-local memory.

However, the performance of highly durable configurations (i.e., small values of *b*) remains poor. So, the extension manager tries another extension. The new extension, *delayed-fsync* (Figure 5), extends the behavior of `fdatasync` so that it waits for the *previous* call to `fdatasync`. This design ensures that the system loses at most 2 updates. The extension also implements a fast-path optimization that extends the kernel to expose a shared variable to userspace that tracks the number of completed `fdatasync` operations on each of a process' open files. The `fdatasync` extension reads from the shared variable and only execute system calls in the event that the previous `fdatasync` has not completed. To use delayed-fsync, the extension manager creates an extension class for `fdatasyc` that has bounded computation and function capabilities to execute `io_uring` and bpftime map helpers for process-kernel shared memory.

6 Evaluation

We implement the 6 case studies on bpftime and answer the following questions:

1. How does bpftime's performance overhead compare to state-of-the-art extension tools, including eBPF, Lua, WebAssembly, ERIM [66], and RLBox [44]?
2. Why does bpftime impose lower runtime overhead than existing tools,?
3. How compatible is bpftime with existing extension use-cases and kernel eBPF runtime?

Experimental Setup. We evaluate bpftime running on two servers. Server A is a dual-socket Intel Xeon Gold 5418Y Processor (24 cores, 2.00 GHz, 45 MB LLC) with 256 GB DDR5 memory. Server B is a dual-socket Intel Xeon E5-2697-v2 processor (48 cores, 2.7 Ghz, 30 MB LLC) with 256 GB DDR3 memory. Unless otherwise mentioned, each metric is the average of 10 trials. We report averages using geometric mean when that is appropriate (e.g., when calculating an average speedup). We compare bpftime to two baselines: the native execution on each system (native) and the performance running the extension on the linux eBPF ecosystem.

6.1 bpftime Performance

This section evaluates bpftime's performance across the 6 case studies. In summary, we find that bpftime offers a significant improvement compared to the current state-of-the-art extension frameworks such as eBPF, Lua, WebAssembly, RLBox [44], and ERIM [66].

6.1.1 Nginx Plugin

We evaluated bpftime performance improvements for extending Nginx against the current plugins approaches, including Nginx's builtin support for Lua and WebAssembly, and prior state-of-the-art systems including ERIM [66] and RLBox [44]. We deployed security and tracing modules for each framework in Nginx on System A and benchmark the system using `wrk` (8 threads, 64 connections, 30s tests, averaged over 20 runs). Figure 6 shows throughput results: bpftime introduces 2% overhead compared to native Nginx. In contrast, Lua and WebAssembly incur 11% and 12%, so bpftime achieves 5.5×

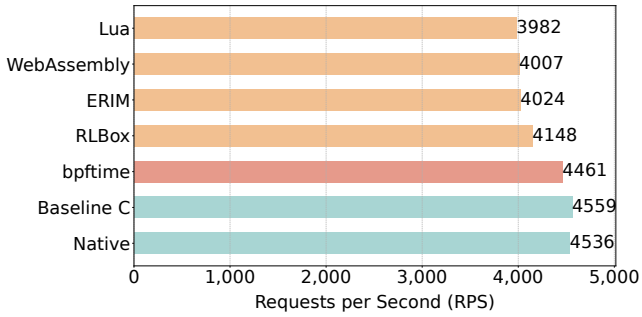
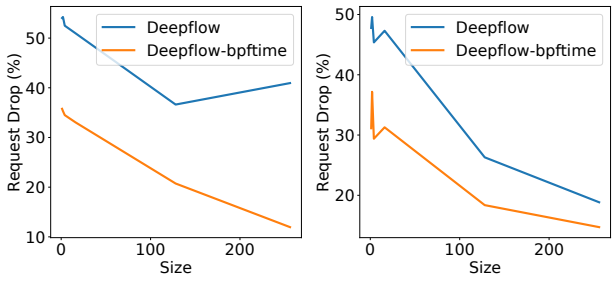


Figure 6: Comparison of extension approaches for Nginx module



(a) HTTP (b) HTTPS

Figure 7: Deepflow throughput overhead

and 6× lower overhead, respectively. ERIM and RLbox impose 11% and 9%, so bpftime achieves 5.5× and 4.5× lower overhead, respectively. In summary, this experiment illuminates a tradeoff between the current extension approaches and bpftime: bpftime and EIM will require more development time to write the specification and pass the verifier when compared to prior work, but significantly improve efficiency and safety.

6.1.2 Deepflow

We deploy Deepflow on System B and use it to monitor a microservice that returns a random string for each request, implemented on a Golang server, and configured to use either HTTP and HTTPS. We route traffic to the service using `wrk`, configured to use 10 threads with 500 concurrent connections for a 10-second test duration. We execute the tests with three configurations: native microservice execution without Deepflow, Deepflow using eBPF, and Deepflow-bpftime using bpftime.

We measure the server's throughput as the response size increases from 1KB to 256KB (Figure 7). Without DeepFlow, the microservice achieves 250,000 requests/s for small responses and 47,000 requests/s for large responses. With DeepFlow using eBPF, throughput drops by up to 54%. bpftime improves DeepFlow throughput by at least a factor of

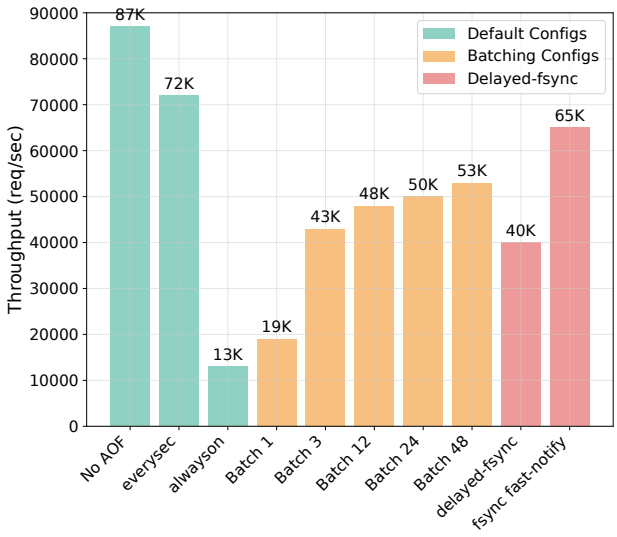


Figure 8: Redis throughput with bpftime Batch-I/O and Delayed-fsync extensions

1.5. DeepFlow's remaining overhead comes from the system's large set of probes that it uses throughout both the kernel and in userspace.

6.1.3 Durability Tuning

We evaluate the Durability Tuning use-case on System A and use redis-bench [65] to send 1M set requests with 5 parallel clients and 3 byte payloads. We execute Redis with the built-in configurations (always on, everysec, and No AOF), varied batch sizes (1, 3, 12, 24, and 48), delayed-fsync, and delayed-fsync with the fast-notify optimization. We configure Redis with always on when configured to use a batching and delayed-fsync. Figure 8 shows the throughput of each of the configurations.

The batched-I/O exposes interesting durability/performance tradeoffs. Using a batch size of 48 improves Redis always on throughput by a factor of 4.17, and incurs a modest durability cost of losing 24 updates in the event of a crash (roughly half of the function in a batch are `writes`). For comparison, the everysec configuration can lose 72,000 updates in the event of an untimely crash. Thus, a batch size of 48 reduces data loss relative to everysec by three orders of magnitude. Additionally, we observe that the batched-I/O implementation improves Redis performance even for small batches: e.g., a batch size of 1 improves Redis always on throughput by a factor of 1.51.

Delayed-fsync presents a powerful configuration. On its own, delayed-fsync achieves a throughput of 40k requests/second, which is 4.15 times larger than the Redis always on's throughput. Adding the fast notify user-kernel interaction further accelerates Redis to 65k request/second, more than 5

Test	Native (s)	bpftime (s)
Passthrough, fstat	3.65	.176
LoggedFS, fstat	7.40	.184
LoggedFS, openat	17.0	0.074
Passthrough, find	5.1	1.6

Table 2: FUSE operation latency.

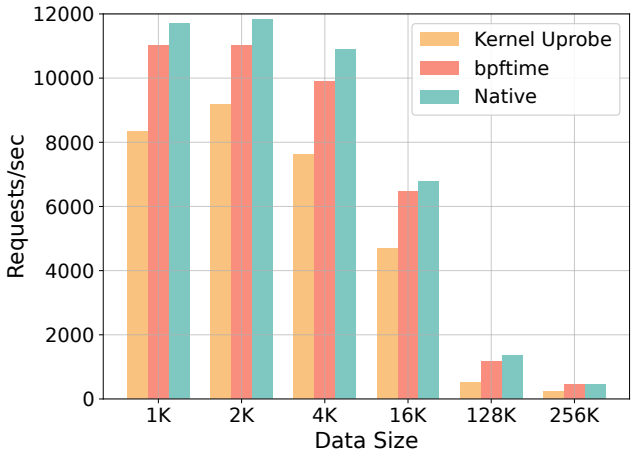


Figure 9: SSlsniff test with nginx

times larger than Redis alwayson. In sum: delayed-fsync with fast notify is only 10% slower than Redis everysec, but it reduces the number of updates that could be lost in a crash by 5 orders of magnitude.

6.1.4 FUSE

We evaluate the performance improvement from using bpftime to implement caching in FUSE when running on System A. We deploy two FUSE file systems: Passthrough, which passes filesystem operations to the underlying file system directly, and LoggedFS [41], which logs all file system operations to a file before passing them to the underlying file system. We measure the end-to-end latency of three workloads: one that issues 100000 `fstatat` calls to a file in the FUSE directory, one that issues 100000 `openat` calls to a file in the FUSE directory, and one that uses the linux utility `find` to travel through the linux 6.7 source code directory. Table 2 shows the results on FUSE and with bpftime caching. We observe that the caching extension implemented with bpftime accelerates the latency of the workloads by a factor of up to 2.4 orders of magnitude.

6.1.5 Sslsniff

We evaluate `sslsniff` on System B while monitoring Nginx. Using the `wrk` benchmark, configured with 4 threads, 512 concurrent connections, and a 10-second test duration. The evaluation includes three configurations: native, `sslsniff` with eBPF, and `sslsniff` with bpftime. We measure nginx throughput when varying response size from 1 KB to 256 KB; figure 9 shows the results. eBPF `sslsniff` significantly impacts Nginx's performance, reducing throughput by up to 28.06% compared to native execution. In contrast, bpftime `sslsniff` has a much smaller effect, with a worst-case throughput reduction of only 7.41

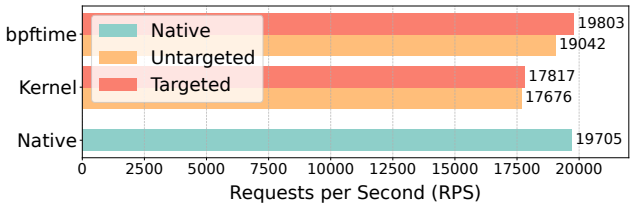


Figure 10: syscount test with nginx

Bench Type	eBPF(ns)	bpftime (ns)
Uprobe	2561.57	190.02
Uretprobe	3019.45	187.10
Syscall Tracepoint	151	232
User memory read	23.3	1.5
User memory write	23.9	1.4
hash_map_update	50.8	23.8
hash_map_delete	19.5	10.1
hash_map_lookup	9.8	22.0

Table 3: Microbenchmark comparison of bpftime and eBPF

6.1.6 Syscount

We evaluate syscount on System B using the same Nginx server config as in the sslsniff evaluation. Figure 10 compares the measured throughput across five configurations. eBPF Syscount reduce throughput on monitored and unmonitored processes by 10.3% and 9.6%, respectively. In contrast, bpftime syscount reduces throughput on the monitored process by 3.36% and does not affect the unmonitored process.

6.2 Microbenchmark Performance

We use microbenchmarks to analyze bpftime's performance.

bpftime vs. eBPF. Table 3 compares bpftime performance to eBPF on microbenchmark operations. First, we create microbenchmarks that measure the overhead of each type of extension entry (`uprobe`, `uretprobe`, or `syscall tracepoint`) on System A. bpftime is more than an order of magnitude faster than eBPF for `uprobe` and `uretprobe`, but about 1.5x slower for `syscall tracepoint`. Then, we measure the latency of using helper functions to access userspace memory and eBPF maps and to update and delete hash entries on System A. We observe that bpftime operations are up to an order of magnitude faster than comparable eBPF operations, since bpftime's extension compiler can inline calls and does not require the address space of kernel eBPF

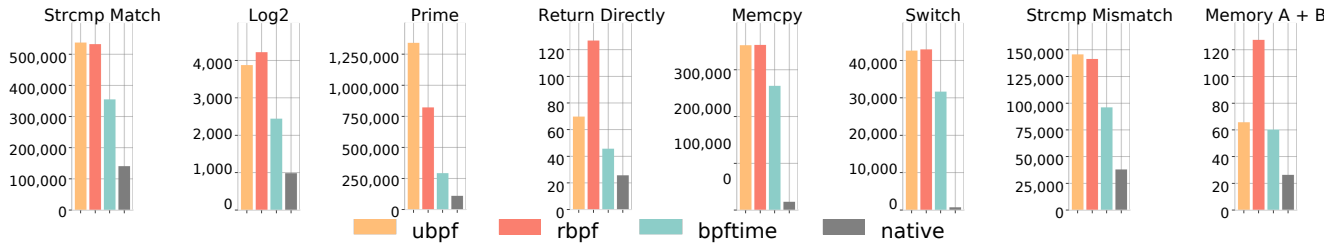


Figure 11: Performance comparison of bpftime, ubpf, rbpf, and native on microbenchmarks.

bpftime execution engine efficiency:. We evaluate the efficiency of bpftime's runtime compared to ubpf [28] and rbpf [28] on 8 micro-benchmarks, including integer heavy computation ('log2', 'prime'), memory heavy workloads ('memory a+b', 'memcpy'), cstring benchmarks ('strcmp match', 'strcmp mismatch'), and control-flow benchmarks ('return directly', 'switch'). Figure 11 shows the latency of each microbenchmark and indicates that bpftime is significantly faster than existing userspace eBPF runtimes. Namely, bpftime accelerates average benchmark latency by a factor of 1.53 and 1.72 compared to ubpf and rbpf, respectively.

bpftime Load latency. We evaluate the latency of loading an extension with bpftime on system A. We write an extension that monitors `malloc` in `libc`, load it into a running process, and observe a 48ms load latency. For comparison, we load the same extension using LD_PRELOAD, and observe a 30ms load latency.

bpftime Cost Breakdown. We isolate the performance improvements of bpftime's two key optimizations: MPK isolation and concealed extension entries. We create a microbenchmark that repeatedly executes a null extension function. The microbenchmark calls the extension function with three settings: first, through its trampoline; second, without the trampoline but with the bpftime execution engine; and third, without the trampoline or execution engine. We observe that the average latency of calling the extension function with the trampoline is 190ns, without the trampoline but with the extension entry is 106ns, and without both is 1.35ns. This means that concealed extension entries save 1.35ns for every unused extension entry that the application executes dynamically; since extension managers often use bpftime's automated extension entries (see §5), these small per entry savings yield large end-to-end performance improvement. The tradeoff is that concealed extension entries require a trampoline that adds an additional 84ns for every executed extension. We reran the experiments with MPK isolation on and off, and did not observe a measurable difference in average extension execution.

6.3 bpftime Compatibility

We evaluate the compatibility of bpftime with existing eBPF extensions. We test 17 BCC [52] and bpftrace [68] tools on bpftime without code changes. Also, bpftime fails only one test in the the bpf-conformance test suite [32], whereas ubpf and rbpf fail 22 and 23, respectively.

7 Related Work

EIM and bpftime improve the safety and efficiency of software extensions in userspace. EIM allows extension manager to specify fine-grained interconnectedness/safety tradeoffs, while bpftime enforces the tradeoffs efficiently. We outline the related work on userspace extensions, kernel extensions, Intra-process Isolation, and Userspace eBPF.

Userspace Software Extensions. Userspace software extensions fall into three categories. Some systems execute extensions in the same execution context as the host application and thus provide efficiency but not safety or isolation, such as dynamic binary instrumentation tools [8, 39, 45] and `LD_PRELOAD`-based tools. Other systems use SFI [69] to provide isolation, but either do not support safety/interconnectedness tradeoffs [23, 38, 75] or lack the ability to specify such tradeoffs at a fine granularity [16, 44]. Moreover, such systems are typically inefficient due to the overhead of SFI's runtime checks. Finally, some systems [7, 11, 31, 37] use subprocess abstractions for isolation. These systems were not designed for extensions and would be inefficient for extension use-cases since they require context-switch-like overheads to switch between contexts. Moreover, some [11, 37] lack support for specifying fine-grained interconnectedness/safety tradeoffs. Others [7, 31] could support fine-grained interconnectedness/safety tradeoffs, but their usage models do not support extensions: a user of these tools would have to modify their application to extend it.

Kernel Extensions. Many systems support kernel extensions [5, 9, 15, 58, 63]; none of these systems support EIM's rich set of fine-grained interconnectedness/safety trradeoffs. Moreover, most [5, 9, 58, 63] use SFI to enforce safety and isolation, which imposes runtime overhead. eBPF has emerged as the de facto extension framework in linux and is increasingly

used in academia [15, 76] and industry [4, 12]. eBPF provides an interface for specifying per hook interconnectedness/safety tradeoffs, but (1) does not support EIM's fine granularity since eBPF bundles large sets of features into sets called program types, (2) does not support all of EIM's features (e.g., eBPF lacks constraints on host-provided functions), and (3) has poor userspace extension support. bpftime's concealed extension entries approach is inspired by recent efforts to support kernel eBPF extensions without traps [30].

Intra-process Isolation. bpftime's use of intra-process memory protection is adopted from the approaches taken by Donky [57], ERIM [66] and libmpk [51]. bpftime's key difference between these systems is its target domain—bpftime applies these ideas to extend an existing application without requiring source-code changes, whereas prior work uses intra-process memory protection to provide lightweight protection domains that a rewritten application can use.

Userspace eBPF. ubpf [28] and rbpf [53] implement virtual machines for userspace eBPF. These systems have high runtime overhead and incomplete eBPF support.

8 Conclusion

In this work, we presented EIM and bpftime, which together provide software extensions that are safer and more efficient. EIM is a new model that allows an administrator to specify fine-grained interconnectedness/safety tradeoffs, effectively reducing the potential for a buggy extension to harm the extended application or underlying system. bpftime is a new extension runtime that enforces EIM specifications. bpftime is more efficient than existing extension frameworks because it uses eBPF-style verification, hardware supported intra-process isolation features, and binary rewriting techniques. Plus, bpftime is easy to adopt since it is compatible with the current eBPF ecosystem. We released bpftime as an open-source project and have cultivated a small user-base. We evaluated EIM and bpftime using 6 use cases to show that they are useful and efficient.

9 Acknowledgements

We would like to thank our shepherd, Ryan Huang, and the anonymous reviewers for their insightful comments and suggestions. This work was supported by the National Science Foundation under grant CNS-2236966.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), nov 2009.

[2] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. Autosteer: Learned query optimization for any sql database. *Proc. VLDB Endow.*, 16(12):3515–3527, August 2023.

[3] BCC Authors. syscount. https://github.com/iovisor/bcc/blob/master/libbpf-tools/syscount.c.

[4] Theophilus A. Benson, Prashanth Kannan, Prankur Gupta, Balasubramanian Madhavan, Kumar Saurabh Arora, Jie Meng, Martin Lau, Abhishek Dhamija, Rajiv Krishnamurthy, Srikanth Sundaresan, Neil Spring, and Ying Zhang. Netedit: An orchestration platform for ebpf network functions at scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 721–734, New York, NY, USA, 2024. Association for Computing Machinery.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 267–283, New York, NY, USA, 1995. Association for Computing Machinery.

[6] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into Reduced-Privilege compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, April 2008. USENIX Association.

[8] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, page 133–144, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 45–58, New York, NY, USA, 2009. Association for Computing Machinery.

[10] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 320–332, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[12] cilium. ebpf-based networking, observability, security solution. https://cilium.io.

[13] Emma Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, August 2020.

[14] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020.

[15] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 249–264, 2024.

[16] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.

[17] The Apache Software Foundation. Developing modules for the apache http server 2.4. https://httpd.apache.org/docs/2.4/developer/modguide.html.

[18] frida. Cross-platform instrumentation and introspection library written in c. https://github.com/frida/frida-gum.

[19] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Cristiano Giuffrida, Călin Iorgulescu, and Andrew S. Tanenbaum. Mutable checkpoint-restart: automating live update for generic server programs. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, page 133–144, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Google. Chrome extensions. https://developer.chrome.com/docs/extensions.

[22] Google Chrome Developers. Manifest - chrome extensions documentation. https://developer.chrome.com/docs/extensions/reference/manifest, 2024. Accessed: 2025-06-02.

[23] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[24] David Hu, Sophie Alpert, and Emily Eisenberg. Vim awesome. https://vimawesome.com/.

[25] F5 Inc. Dynamic modules | nginx documentaiton. https://docs.nginx.com/nginx/admin-guide/dynamic-modules/dynamic-modules/.

[26] Kong Inc. Load balancing - kong. https://docs.konghq.com/gateway/latest/get-started/load-balancing/.

[27] Kong Inc. Prometheus - plugin - kong docs. https://docs.konghq.com/hub/kong-inc/prometheus/.

[28] iovisor. Userspace ebpf vm. https://github.com/iovisor/ubpf.

[29] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

[30] Jinghao Jia, Michael V. Le, Salman Ahmed, Dan Williams, Hani Jamjoom, and Tianyin Xu. Fast (trapless) kernel probes everywhere. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 379–386, Santa Clara, CA, July 2024. USENIX Association.

[31] Yuzhuo Jing and Peng Huang. Operating system support for safe and efficient auxiliary execution. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 633–648, Carlsbad, CA, July 2022. USENIX Association.

[32] Alan Jowett. Cross-platform instrumentation and introspection library written in c. https://github.com/Alan-Jowett/bpf_conformance/.

[33] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant {OS} updates via userspace {Checkpoint-and-Restart}. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 605–619, 2016.

[34] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *11th International Software Product Line Conference (SPLC 2007)*, pages 223–232, 2007.

[35] Linux kernel maintainers. Bpf kernel functions (kfuncs). https://docs.kernel.org/bpf/kfuncs.html.

[36] libcapstone Authors. libcapstone is a disassembly library. https://github.com/libcapstone/libcapstone.

[37] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, Savannah, GA, November 2016. USENIX Association.

[38] LuaLab. Lua the programming language. https://www.lua.org/home.html.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI05*, pages 190–200, Chicago, IL, June 2005.

[40] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Biswajit Mazumder and Jason O. Hallstrom. A fast, lightweight, and reliable file system for wireless sensor networks. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016.

[42] MITRE. Cve-2024-31449: Redis lua script stack overflow. Common Vulnerabilities and Exposures, 2024.

[43] Mozilla. Customize firefox. https://addons.mozilla.org/en-US/developers/.

[44] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.

[45] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI07*, San Diego, CA, June 2007.

[46] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *12th Annual Network and Distributed System Security Symposium*, San Diego, California, 2005.

[47] NVD. Cve-2021-44790. https://nvd.nist.gov/vuln/detail/CVE-2021-44790.

[48] Inc. OpenResty. Openresty. https://openresty.org/en/.

[49] Oracle. Mysql. https://www.mysql.com/.

[50] PostGIS PSC & OSGeo. About postgis. https://postgis.net/.

[51] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[52] IO Visor Project. Bpf compiler collection (bcc), 2023. Available: https://github.com/iovisor/bcc.

[53] qmonnet. Rust virtual machine and jit compiler for ebpf programs. https://github.com/qmonnet/rbpf.

[54] Red Hat. What is selinux? https://www.redhat.com/en/topics/linux/what-is-selinux, 2024. Accessed: 2025-06-02.

[55] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[56] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, Boston, MA, August 2022. USENIX Association.

[57] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys–efficient {In-Process} isolation for {RISC-V} and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694, 2020.

[58] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*, Seattle, WA, October 1996. USENIX Association.

[59] Avthar Sewrathan and Bryan Clark. Top 8 postgresql extensions. https://www.timescale.com/blog/top-8-postgresql-extensions/.

[60] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In David Kotz and John Wilkes, editors, *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*, pages 170–185. ACM, 1999.

[61] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 420–437, 2023.

[62] Asia Slowinski, Traian Stancescu, and Herbert Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 125–137, Boston, MA, June 2012. USENIX Association.

[63] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 207–222, New York, NY, USA, 2003. Association for Computing Machinery.

[64] The Emacs Community Taiwan. awesome-emacs: A community driven list of useful emacs packages, libraries and other items. https://github.com/emacs-tw/awesome-emacs.

[65] Redis Team. Redis benchmark. https://redis.io/docs/management/optimization/benchmarks/.

[66] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({{{{{MPK}}}}}). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[67] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, February 2017. USENIX Association.

[68] IO Visor. bpftrace: High-level tracing language for linux ebpf. GitHub repository, 2023. https://github.com/iovisor/bpftrace.

[69] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[70] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 133–147, New York, NY, USA, 2020. Association for Computing Machinery.

[71] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[72] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2989–3002, 2022.

[73] XRay. Resolving bilibili's major site incident with openresty xray. https://tinyurl.com/4ten75h2.

[74] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 293–300, 2023.

[75] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[76] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[77] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 142–157, New York, NY, USA, 2023. Association for Computing Machinery.



Tintin: A Unified Hardware Performance Profiling Infrastructure to Uncover and Manage Uncertainty

Ao Li Marion Sudvarg Zihan Li Sanjoy Baruah Chris Gill Ning Zhang
Washington University in St. Louis

Abstract

Hardware performance counters (HPCs) enable the measurement of microarchitectural events, which are crucial for tracking and predicting program behavior. High-fidelity measurement and precise attribution are essential for accurate profiling. However, existing profiling tools have fundamental challenges in both aspects. In measurement, numerous events compete for limited hardware monitoring resources; while for attribution, applications have diverse requirements, but systems provide limited support. Existing tools mitigate the former limitation through event multiplexing, but this approach introduces non-trivial errors. The latter limitation, however, remains largely unaddressed.

This paper introduces *Tintin*, an HPC profiling infrastructure with a modular three-component design that addresses both challenges. Tintin introduces mechanisms to mitigate multiplexing errors by characterizing uncertainty at runtime, scheduling events to minimize it, and reporting uncertainty to applications. It also proposes the Event Profiling Context (*ePX*) as a new OS primitive to unify diverse profiling requirements. Tintin is evaluated using benchmarks as well as real-world resource orchestration, performance debugging, and intrusion detection systems, to demonstrate its ability to improve hardware profiling with low runtime overhead.

1 Introduction

Performance profiling involves *measuring* runtime cost metrics and *attributing* these measurements to a portion of one or more programs' execution. Modern processors provide hardware performance counters (HPCs) that can profile microarchitectural events (e.g., cache loads and misses). The hardware events collected from HPCs, referred to as HPC data, are valuable across a wide range of application domains, including debugging [5,23,49,50], workload optimization [8,15,35], power analysis [58,60,76], diagnostics [8], online resource provisioning [25,27,37,54,72], and intrusion detection [19,20]. However, existing profiling infrastructures

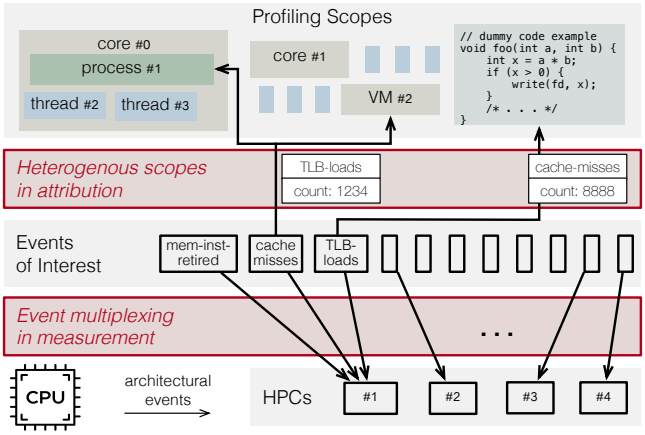


Figure 1: An event profiling infrastructure must accurately measure and attribute microarchitectural events to a heterogeneous set of profiling scopes, while managing a limited number of HPCs.

for hardware events have fundamental limitations in both measurement and attribution.

Problems. When measuring HPC data, each individual HPC can be programmed to record only a single type of event at a time, but the number of events of interest (typically tens to hundreds) often exceeds the available HPCs (usually 2–6), as shown in Figure 1. Therefore, many existing solutions [44,46] adopt *event multiplexing*, scheduling events in a time-shared manner (where each event typically receives an equal portion of time). Observed counts are interpolated to estimate the total; however, this multiplexing inevitably introduces errors, as events may remain unmonitored for extended intervals. Most approaches to avoiding multiplexing involve running the target application multiple times, each time monitoring a different subset of events, and then merging the results into a single trace [31,41,48]. However, these offline techniques are impractical for emerging applications that use HPC data as real-time input for online decision-making [25,35,54,72], such as dynamic resource orchestration.

Furthermore, hardware event attribution remains challeng-

ing. Unlike software events [6] (e.g., memory usage and page faults) that are directly tied to profiled program execution, hardware events are counted independently by standalone HPCs. Attribution requires system support to align these events with program execution. However, there is a fundamental mismatch between the abstractions provided by existing tools [10, 33, 53, 55, 73] and application requirements. Existing tools only allow profiling events for specific tasks (i.e., threads/processes) or cores. For task-level profiling, HPCs start counting when the target task is scheduled in, then stop when it is switched out. While this design is straightforward to implement – in Linux, events are bound to the task_struct and logic is added to the CPU rescheduling routine – it means the profiling scope[1] is bound to schedulable entities (e.g., task/core switches). However, there is a significant need to define more flexible profiling scopes in practice. For example, developers may want to profile events for specific code regions [1, 4, 35, 62, 64] as shown in Figure 1. Current abstractions are insufficient for these heterogeneous profiling requirements, leading to event misattribution. This heterogeneity might also cause conflicts between overlapping scopes that result in some events remaining unmeasured.

Our Solution – Tintin. This paper proposes Tintin, a new hardware event profiling infrastructure that harnesses two key ideas to address the aforementioned challenges. First, Tintin characterizes multiplexing errors as uncertainty at runtime. Leveraging this, Tintin implements an event scheduling algorithm to minimize the overall uncertainty. The algorithm is grounded in real-time scheduling theory and is proven to be optimal. Since uncertainty cannot be fully eliminated, Tintin also reports it to the application via user-space interfaces; to the best of our knowledge, this is the first profiling tool to do so. We present case studies that demonstrate how these reported values may improve application-level decision making. Second, to address the inflexibility of attribution, Tintin proposes to elevate profiling scopes via a first-class object, the Event Profiling Context (*ePX*). Heterogeneous *ePX*s are uniformly managed across the entire system, allowing for flexible scope definition while resolving conflicts between overlapping scopes by jointly scheduling their events.

The system design of Tintin consists of three modular kernel components, each introducing techniques to improve accuracy and reduce overhead in realizing the basic design. *Tintin-Monitor* simultaneously characterizes uncertainty while measuring event counts. As ground truth is not available at runtime, it estimates uncertainty using variance as a proxy. To reduce overhead, it adopts incremental variance updates. *Tintin-Scheduler* represents event multiplexing as an elastic real-time scheduling problem [12, 13] to minimize total expected uncertainty. It presents a novel extension of elastic scheduling to multiple HPCs, assigning shares of HPC time and constructing a schedule in quasilinear time. *Tintin-Manager* serves as another level of indirection to uniformly handle and manage the heterogeneous attribution requirements originating from user space. It extends Linux's familiar perf_event API to specify explicit profiling scopes with flexible granularity and report uncertainty back to user space.

Summary of Results. Tintin is evaluated on three real-world applications: Pond [37] for cloud resource provisioning, DMon [35] for performance debugging, and the Diamorphine rootkit [43] for intrusion detection. In these case studies, we demonstrate that Tintin can be easily integrated into existing applications in place of other profiling tools with minimal adaptation. By tracking, minimizing, and reporting uncertainty, Tintin allows Pond to predict latency sensitivity with 64% greater accuracy, and the AUC for Diamorphine rootkit classification increases by 22.8%. Additionally, Tintin's operability to adjust targeted profiling scope allows DMon to identify performance issues in a push-button fashion. Furthermore, we conduct a comprehensive evaluation to assess accuracy improvements, runtime overhead, and scalability on SPEC 2017 [11] and PARSEC [9] benchmarks. Results indicate that Tintin incurs low overhead (2.4%) while interpolating multiplexed event counts 3.09× more accurately than the state of the art.

Contributions. This paper proposes Tintin, a kernel infrastructure for hardware event profiling. It includes:

- An uncertainty-driven hardware performance measurement mechanism that quantifies, reports, and schedules to minimize estimated errors.
- A new OS primitive, the Event Profiling Context, to enable flexible and accurate attribution while resolving compatibility issues between conflicting profiling scopes.
- The design, implementation, and evaluation of Tintin[2], a kernel hardware event profiling infrastructure with a three-component modular structure.

2 Background

Performance Monitoring Counters. Modern processors make hardware event profiling available to system software via a per-core Performance Monitoring Unit (PMU), which provides a set of programmable HPCs that can be configured individually to measure a specific type of microarchitectural event, e.g., cache or bus accesses, cache writebacks or refills, branch misprediction, etc. When enabled, an HPC increments whenever its programmed event occurs. The number of measurable events is substantial, typically exceeding several

[1] Here, we adopt definitions from Saltzer and Schroeder's subject-object model [56, 57]. Hardware events are the *objects* of profiling, while the corresponding sources of those events are the *subjects*. A "scope" therefore defines a subject to profile. This could be execution on CPU core(s), across a virtual machine, a single thread, or originating from a specific code segment.

[2] Source code is available at: https://github.com/WUSTL-CSPL/tintin-kernel and https://github.com/WUSTL-CSPL/tintin-user.

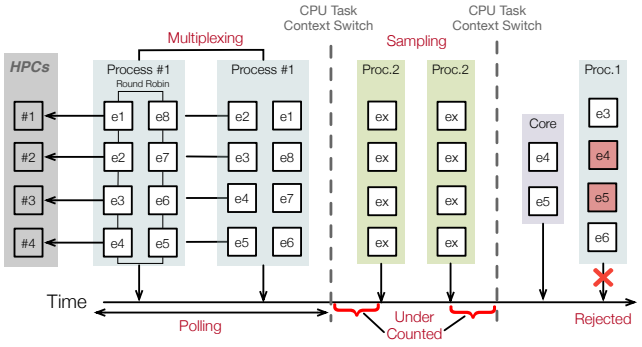


Figure 2: The Linux perf_event subsystem multiplexes events on limited HPCs. It is not aware of scope overlap; joint scheduling and common attribution remain unsupported. If a per-core event is pinned to an HPC, per-task events may be rejected, leading to measurement starvation. perf_event supports both instrumented polling and time- or event-triggered sampling. When sampling, events may be undercounted when the sampling intervals do not align with task context switches.

dozen on ARM processors (e.g., 58 on the Cortex-A53 [38] and 151 on the Cortex-A78 [39]) and over one thousand on Intel processors (e.g., 1,623 on HaswellX [75]).

2.1 Linux perf_event Subsystem

The Linux perf_event subsystem [65] is Linux's kernel-level abstraction layer for interacting with HPCs. It serves as the de facto infrastructure widely utilized by many tools such as PAPI [10], Intel EMON [33], VTune [55], pmu-tools [73], and the Linux Perf utility [70]. It provides access to both hardware-level HPC data and software-level data (e.g., memory footprint and tracepoints). This section discusses its hardware aspects, which are the exclusive focus of Tintin.

Polling vs. Sampling. Events on HPCs can be monitored through either *polling* [44,62] or *sampling* [21,74]. In polling, HPCs are configured and enabled at explicit points during program execution (via instrumented syscalls) and then read at later points. With sampling, HPCs are read periodically, either in time-triggered fashion at periodic intervals (e.g., by explicit timer handlers or in the system timer interrupt routine) or in event-triggered fashion every time a pre-defined threshold is reached (e.g., using Intel's Precise Event-Based Sampling (PEBS) hardware or ARM's PMU interrupt). It is worth noting that sampling does not necessarily incur less overhead than polling. They in fact represent a trade-off between intrusiveness and precision. Polling typically requires instrumentation but offers higher accuracy by placing calipers at the boundaries of profiling scopes. In contrast, sampling does not require instrumentation but suffers from misattribution, as it does not align events precisely with the target scope, such as task context switches as shown in Figure 2.

Event Scheduling. When an HPC is configured to count a particular event, we say that the event is *scheduled* on the counter. If the number of events to be monitored exceeds the number of available HPCs, they must time-share the counters, being multiplexed in a round-robin fashion. Figure 2 presents an illustrative example in which the available HPCs on the CPU are limited to 4, while Process #1 requires monitoring for 8 events. In this scenario, the events {e1, e2, e3, e4} are allocated to the initial time slice, followed by the scheduling of events {e2, e3, e4, e5} in the subsequent time slice.

Profiling Scope. The perf_event subsystem enables specification of hardware event monitoring for either individual tasks (processes/threads) or CPU cores. A key advantage of this design is its simplicity as the kernel only needs to track specified events and their counts within existing data structures for the profiled subject, e.g., the task_struct. It additionally requires minimal additional logic in the task scheduler to switch between the events associated with different tasks or cores.

Upon CPU task scheduling, perf_event first schedules events bound to the current core before adding those associated with the active process. Since events are bound to per-core and per-task data structures, they are managed independently, even if they share common events of interest. The right side of Figure 2 illustrates this scenario: a user assigns two events, {e4, e5}, to the current core; these are placed on HPCs #2 and #3. Consequently, events {e4, e5} monitored for Process #1 are not scheduled, even though they represent the same event types, resulting in unnecessary starvation.

2.2 Sources of Uncertainty in HPC Profiling

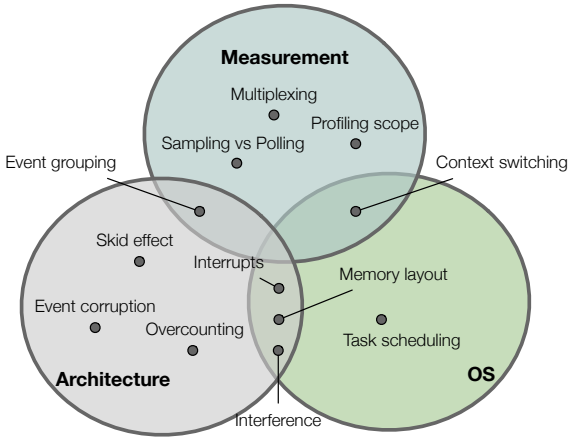


Figure 3: Uncertainty sources.

Sources of HPC measurement uncertainty can originate from a combination of three factors: architectural uncertainty, the operating system, and measurement method limitations, as illustrated in Figure 3.

Architectural Uncertainty. This type of uncertainty is caused by flaws in the microarchitecture. Several known sources of such uncertainty have been studied in prior work.

- Skid effect [28, 74]: To support event-triggered sampling, many PMUs can be configured to deliver an interrupt when an HPC overflows. On some processors, additional instructions execute during the time delay between interrupt initiation and signal delivery to the CPU; associated events may be incorrectly counted or attributed.
- Event corruption [21]: On some Intel processors with Hyper-Threading, certain event counts will be corrupted if monitored by matching counters on the same physical core.
- Overcounting [67, 68]: Certain events will cause an unrelated counter to increment on several older x86 processors from both Intel and AMD.

OS Uncertainty. Profiling might not yield identical results across different runs of the same program, primarily because the runtime environment can vary between measurements. For example, factors such as OS activity, program scheduling in multitasking environments, memory layout and pressure, and multi-processor interactions may differ from run to run.

Measurement Method Uncertainty. Many tools have been developed to facilitate the collection of HPC data through various measurement methods. However, these methods often introduce uncertainty. For example, as discussed in the previous section, the Linux perf_event subsystem provides profiling support through several mechanisms, each contributing to potential measurement error. These include the data collection technique (e.g., polling or sampling), the profiling scope (e.g., thread, process, core, or multiple cores), and the use of multiplexing. As polling and sampling each represent a different trade-off between accuracy and intrusiveness, removing their uncertainty is out of Tintin's scope. In fact, Tintin works seamlessly with either technique.

3 Challenges and Pitfalls

Limited HPCs Compared to Events of Interest. Modern CPUs support dozens to hundreds of types of events that can be monitored. However, the number of available HPCs is very limited; most processors provide only 2–6 HPCs per physical core [70, 75], and this number is effectively halved per logical core when Simultaneous Multithreading (SMT) is enabled. Additionally, Linux reserves one HPC by default for the watchdog timer, further reducing the counters available. This creates a fundamental tension between the number of events of interest and the available HPCs.

Can We Profile Fewer Events? Existing work has sought to select relevant events carefully for a given application. However, the number of events of interest remains large. For example, in [45], the 120 events available on an ARMv7-A CPU were narrowed to just 34 of importance for predictive DVFS. CounterMiner [41], an offline analysis tool for predictive modeling with event counts, achieves the most accurate IPC predictions for HiBench [32] workloads using ~150 events. Furthermore, some applications profile derived metrics, such as

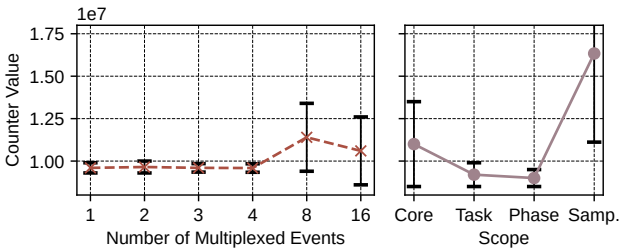


Figure 4: Preliminary examination of multiplexing and scope-based errors. (Left) When the number of events exceeds the number of HPCs, variation in reported event counts increases. (Right) A precise definition of profiling scopes is also essential for accurate results.

Memory_Bound [35]. These combine as many as 16 individual events [18]. Pond [37], which we evaluate as a case study in §8.1.1, provides a memory pooling model for cloud infrastructure. Its model takes 7 derived metrics as inputs, spanning 20 events on Intel Skylake.

Event Multiplexing Introduces Errors. The current approach to managing this limitation is through event *multiplexing*. In the Linux perf_event subsystem, events are scheduled for monitoring in a time-shared fashion on the available HPCs [2, 21, 47, 75]. HPC measurements for each event can then be interpolated to estimate the total count. However, this method unavoidably introduces errors, as the reported values are based on interpolation, and variance in event arrival further contributes to inaccuracies in the results.

Examination of Multiplexing Error. To illustrate this effect, we use the Linux Perf utility to profile the 541.leela_r Go engine in the SPEC CPU®2017 benchmark suite [11] on an Intel Xeon Gold 6130 Skylake CPU. The target PMU has 4 generic counters, so we profiled 1–4 (non-multiplexing), 8, and 16 selected events. The multiplexing frequency in the experiments is set to the default value (4 ms) in Linux perf_event, which aligns with the CPU task scheduling frequency. The mean and standard deviation of reported LLC-load-misses over 10 runs are plotted in Figure 4. Without multiplexing, reported event counts remain stable; they do not vary significantly with the number of events profiled. However, when the number of events exceeds the available counters, the reported metrics vary significantly, as multiplexing fails to capture the true characteristics of the workload [59, 75].

Heterogeneous Profiling Scopes. HPCs are simply counters and are decoupled from the profiling scope. Therefore, profiling infrastructure must align recorded events with program execution on the CPU. Developing such mechanisms is challenging due to the heterogeneity of profiling requirements.

Incomplete Handling of Diverse Scopes. Applications require event attribution to diverse scopes, such as execution instances [25, 37, 54] (e.g., a process, a core, a VM) or code segments [1, 35, 62, 64] (e.g., functions, basic blocks). Existing abstractions only explicitly bind events to per-process or

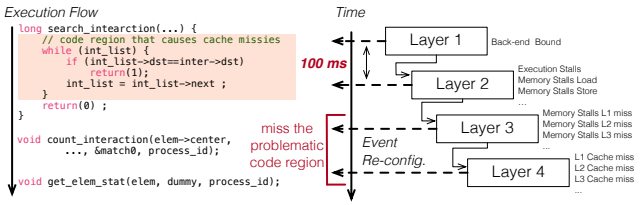


Figure 5: In DMon [35], inflexible profiling scopes cause event misattribution and thus misidentification of data locality issues.

per-core scopes. While the sampling-based method can record program counters (i.e., code addresses) and map events back to source code using debugging tables, this approach cannot filter events for a specific scope and suffers from sampling bias, favoring frequently executed code [16].

Ignorance about Overlapping Scopes. Different scopes may overlap both temporally (e.g., when a process and the core it runs on both have active profiling scopes) and in their profiled event types. If these scopes are unaware of each other, the direct result is that, due to the limited availability of HPCs, events assigned earlier are scheduled, while subsequent assignments are rejected, leading to unfair HPC sharing.

Inflexibility Leads to Misattribution. Without flexible abstractions for defining profiling scopes, users risk mismatches among them; this may lead to misattribution of unrelated event counts. For example, the survey study in [19] examines various applications that aim to characterize program workloads. These often profile the entire task, starting immediately after the process is created, which incidentally captures events from process initialization that are unrelated to the actual profiling scope of interest. To profile events across multiple tasks, dCat [72] pins target tasks to specific cores and then profiles these cores. This approach inevitably includes event counts from context switches on these cores and may also capture events from other tasks running on the same cores.

Misattribution Example. We use Dmon [35] as an illustrative example of the necessity of flexible profiling scopes. DMon [35] is a performance diagnostic tool that detects software data locality issues using memory- and cache-based event profiling. It employs Intel's Top-Down methodology [73] to categorize events into a four-layer hierarchy, monitoring events at subsequent layers based on profiling results from the current layer. As shown in Figure 5, DMon starts by profiling the derived Back-end Bound metric (revealing how often micro-ops are not delivered due to back-end resource shortages) for 100ms. If the program is identified to be back-end bound, DMon then profiles the metrics in the next layer (e.g., execution and memory stall load).

Problematic code segments with data locality issues are often loops that repeatedly perform data operations, e.g., recursive pointer chasing [35]. To identify these operations, DMon must attribute events to the corresponding regions of code. However, DMon modifies events on the fly as it moves through the top-down hierarchy, during which time the target program's execution may move away from the problematic code segments; events from subsequent layers of the hierarchy might therefore be attributed to other segments, with problematic ones remaining undetected. [35] suggests mitigating this by switching to the next layer more swiftly. However, this requires manual tuning of the time window, which varies across different workloads and platforms.

Examination of Mismatched Scope Errors. We extend our preliminary study to quantify the effects of misattribution due to mismatched profiling scopes. We again profile the 541.leela_r benchmark, pinning the workload to a single core and using Linux Perf to count all occurrences of LLC-load-misses in four different profiling scopes: (i) on that core, (ii) by the process, (iii) between explicit polling points instrumented at the beginning and end of program execution, and (iv) using time-triggered sampling. Results are measured across 10 runs and plotted in Figure 4. The instrumented scope produces the most stable results, acting as calipers to filter just those events related to the program.

4 Tintin Design Overview

Tintin is a hardware event monitoring infrastructure that aims to solve these limitations, harnessing the following insights.

4.1 Key Insights

I1. Multiplexing errors can be characterized at runtime and reported to applications.

While multiplexing errors cannot be entirely eliminated, the profiling infrastructure can quantify the expected magnitude of these errors based on observed variance in event rates. These can be reported back to user-space applications alongside the measured counts. These errors serve as indicators of confidence in the measurements and can inform the application's decision-making accordingly.

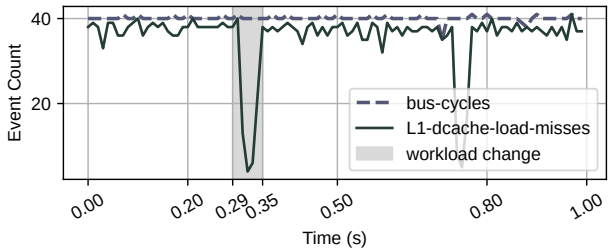


Figure 6: Event count changes over time.

I2. Events exhibit dynamic and distinct errors at runtime, presenting opportunities for scheduling.

By allocating more HPC time to events with larger variance, the overall error can be mitigated. Figure 6 plots counts of the

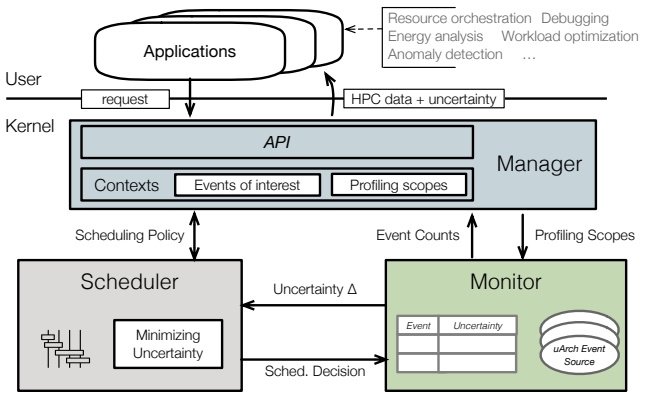


Figure 7: Tintin design overview.

bus-cycles and L1-dcache-load-misses events when profiling the 541.leela_r benchmark for 1 second. This illustrates that variance may depend on the specific execution phase. In this period, L1-dcache-load-misses fluctuates significantly, while bus-cycles remains stable. Allocating more HPC time to the former could reduce measurement errors.

It is worth noting that Tintin does *not* require offline analysis to characterize execution phases of the target workload. Instead, it detects sharp changes online, which is made possible because these typically persist for at least tens of milliseconds [59], as shown by the gray area in Figure 6. Sharp changes at the millisecond level tend to be imperceptible to system performance [25, 45, 54, 72]. Nonetheless, Tintin supports programmable intervals of 1ms or less with negligible overhead, allowing it to capture even these transient effects.

I3. A level of indirection can provide a uniform mechanism to handle the heterogeneity of profiling requirements.

> *"We can solve any problem by introducing an extra level of indirection."* – often attributed to Butler Lampson, who attributes it to David Wheeler.

The two issues related to profiling scope – inflexibility in definition and conflicts due to overlap – both point to the same solution: a new layer to standardize and manage heterogeneous profiling requirements. With such a layer in the kernel, the system can provide a rich programming API to define diverse profiling scopes without adding management complexity. The heterogeneous requirements are translated into a uniform abstraction within the indirection layer. This allows for collaborative management of all profiling requirements across the system, resolving scope conflicts.

4.2 Tintin Structure

Figure 7 depicts the structure of Tintin, which consists of three internal modular components.

Tintin-Monitor (§5) measures HPC data and characterizes errors simultaneously at runtime. It tracks event counts and unmonitored time, and based on this data, employs a lightweight calculation method to estimate errors caused by multiplexing. These error estimates are maintained alongside the HPC data and reported to user-space applications upon request.

Tintin-Scheduler (§6) uses the errors reported by Tintin-Monitor to schedule events on HPCs, assigning each event a unique share of time with the objective of minimizing overall error. This problem is shown to be semantically equivalent to elastic scheduling in real-time scheduling theory [12, 13]. We present a quasilinear algorithm to solve this optimally for events on multiple HPCs, maintaining low overheads.

Tintin-Manager (§7) provides another level of indirection in the kernel. It proposes a new first-class OS abstraction object – the *ePX*– translating heterogeneous profiling requirements from user space into a uniform representation. It then manages these *ePX*s collectively for all applications, avoiding inefficiency caused by conflicting profiling scopes. Additionally, most of Tintin-Manager's APIs have counterparts in existing profiling tools, ensuring minimal effort to migrate existing applications to use Tintin.

5 Tintin-Monitor

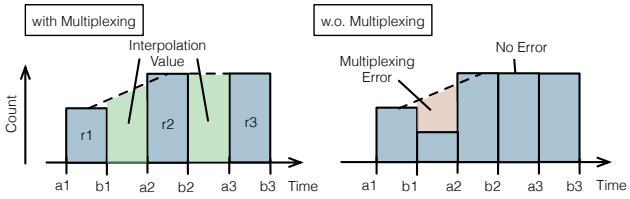


Figure 8: Tintin leverages variance as a proxy to model the uncertainty in the context of multiplexing.

Tintin-Monitor interpolates multiplexed event counts and aims to characterize the resulting estimate errors. These errors are then used for event scheduling and are reported to user space to inform the measurement confidence.

Modeling Runtime Uncertainty. A challenge in calculating runtime errors in estimated event counts is the lack of ground truth measurements during runtime. To address this, Tintin-Monitor leverages measurement variance as a proxy for error. The underlying idea is that existing interpolation mechanisms [47] implicitly assume linear changes in event rates [22] between subsequent counts. Figure 8 shows an example of interpolation, where the rectangles represent the time slices during which events are measured. If the rate of change in interstitial time slices remains strictly linear, there would be no errors in the reported counts. Prior work [22] therefore links interpolation uncertainty to non-linearity in the unmeasured quantities.

Interpolation. Formally, an event $e_i \sim (x_i, \sigma_i)$ has an estimated count x_i and expected uncertainty σ_i due to interpola-

tion over multiplexed observations. As shown in Figure 8, the event e_i is measured with count x_i^j over some continuous time interval $I_i^j = [a_i^j, b_i^j]$ of duration δ_i^j. Then its average rate r_i^j over this interval is x_i^j/δ_i^j. To interpolate over unmonitored intervals, Tintin-Monitor uses a custom implementation of the trapezoid area method (TAM), which was shown to be the most accurate methodology for online interpolation of event counts [47]. Tintin-Monitor assumes a linear rate of change in the event rate, constructing a trapezoid so that its top passes through the midpoint $((b_i^{j+1} + a_i^{j+1})/2, r_i^{j+1})$ of the second measured interval.

Uncertainty. For an event e_i with interpolated count x_i, Tintin-Monitor defines uncertainty as the expected error σ_i in the count. The instantaneous rate of event arrival is a random variable $\mathbf{r}_i$, with a mean rate r_i^j obtained during each measurement interval I_i^j. Since instantaneous rates cannot be measured at every instant, Tintin-Monitor instead characterizes expected variance $V(\mathbf{r}_i)$ by taking the variance of the sample means, weighted by duration. In many stochastic processes, variance scales linearly with time [26]; since we are measuring variance in *rate*, it therefore follows that the expected variance $V(x_i, t)$ in event *count* x_i over the time t that the event is not monitored can be expressed as $V(x_i, t) = V(\mathbf{r}_i) \cdot t^2$. Using variance as a proxy for uncertainty, the expected error σ_i in the estimated event count is $\sqrt{V(x_i, t)}$.

Efficient Incremental Update of Uncertainty. Tintin-Monitor needs to update variance at each monitoring interval. The standard computation of variance requires revisiting all past data points, which is time- and space-consuming. Tintin-Monitor instead implements a weighted version of Welford's method [69] for incremental variance updates. At the end of monitoring intervals I_i^j $(j>1)$, variance is updated as $V_i \leftarrow V_{i-1} \cdot (r_i - \mu_i) \cdot (r - \mu_{i-1})$, where μ is the time-weighted mean over collected counts.

Uncertainty Usage Models. To the best of our knowledge, Tintin is the first hardware event profiling system to report measurement uncertainty to user space. By serving as an indicator of measurement confidence, uncertainty can be explicitly incorporated into application decision-making. While the use case depends on the specific application, our investigation shows that profiling-informed decision making generally follows two broad patterns: rule-based processes [8, 15, 25, 27, 35, 72], which rely on procedural logic derived from human domain expertise; and model-based decision making [20, 37, 54] based on data-driven learning.

For rule-based approaches, conditional logic may be predicated on observed HPC data, e.g., an event count must exceed a certain threshold for an action to be taken. In this context, uncertainty can be incorporated to ensure actions are taken only when the observed data reflect with sufficient confidence that the target condition has been met (i.e., uncertainty is sufficiently low), thereby increasing the reliability of the decision. For example, Caladan [25], a global task scheduler for multicore and multiprocessor systems, attributes memory bandwidth usage to scheduled tasks based on LLC misses and revokes a core from the task with the highest count. However, uncertainty in the profiling data can lead to false identification of high bandwidth usage. By incorporating an uncertainty threshold as an additional condition, the system can avoid mistakenly penalizing tasks.

Model-based approaches can incorporate reported uncertainty values directly into the model as additional input dimensions, allowing the system to account for the confidence of each measurement during prediction. For instance, in our case studies (§8.1), we included error estimates as input features to the predictive models used by Pond to improve the accuracy and robustness of resource orchestration.

Implementation. Tintin-Monitor is activated using the Linux kernel's hrtimer. It leverages the existing hardware interfaces in perf_event for reading and configuring PMUs across architectures, intercepting functions to read raw HPC data to perform TAM-based interpolation and incorporate uncertainty characterization. Variance is initialized by a warm-up period where events are scheduled in a round robin fashion, then updated with each HPC read using Welford's method. To avoid involving floating-point operations in the involved division steps, we apply a scaling factor, where necessary, to the numerator then perform integer division to achieve a fixed-precision result. To prevent overflow, we use 64-bit integers and hand-tune the order of operations to avoid large values.

6 Tintin-Scheduler

Tintin-Scheduler draws from a branch of real-time systems theory that addresses scheduling tasks on limited resources. More specifically, elastic scheduling theory [12, 13] adapts the allocation of processor resources to tasks on overloaded systems to optimize some objective (e.g., result quality). Tintin-Scheduler's goal is similar: it aims to adjust allocations of time for events on limited monitoring resources (HPCs) to minimize overall uncertainty. In this section, we formulate the event scheduling problem and translate it into a form that is semantically equivalent to elastic scheduling.

Scheduling Model. Formally, for a profiling scope, we define $\mathbf{E} = \{e_i\}$, the set of events to be monitored, where $|\mathbf{E}| = n$. Similarly, $\mathbf{C} = \{c_j\}$ denotes the set of HPCs available to monitor them, where $|\mathbf{C}| = m$. The problem is to determine a schedule $\mathcal{S}(t) : \mathbf{C} \rightarrow \{\mathbf{E}, \phi\}$ that defines, at each time t, the event assigned to each counter. We define an event's *utilization* U_i as the fraction of time it occupies a counter. A function $\mathcal{L}_i(U_i)$ characterizes the loss associated with multiplexing event e_i. $\mathcal{L}_i(1)$ is assumed to be 0 (if the event is always counted, there is no multiplexing error), and the function is assumed to be monotonically non-increasing.

Scheduling Policy. From §5, the expected error σ_i in the

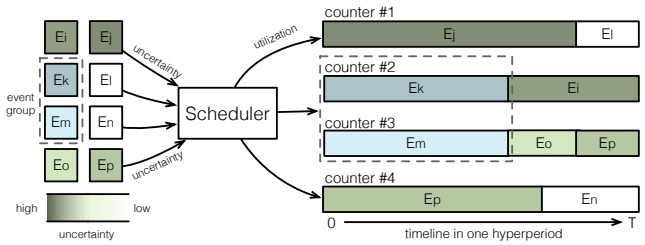


Figure 9: Tintin-Scheduler workflow.

estimated count of event e_i is $\sqrt{V(\mathbf{r}_i)} \cdot t$, where t is the amount of unmonitored time. Future unmonitored time for event e_i is thus proportional to $1 - U_i$, where U_i is its utilization; thus, we express the expected error, normalized to the count, as $\sigma_i = \left(\sqrt{V(\mathbf{r}_k)}/x\right) \cdot (1 - U_i)$. To minimize the sum of squared errors, the loss function $\mathcal{L}_i(U_i)$ is formulated as $\sigma_i^2 = \left(V(\mathbf{r}_i)/x_i^2\right) \cdot (1 - U_i)^2$. The scheduling problem is therefore stated as the following constrained optimization problem:

$$\min_{U_i} \quad \sum_{i=1}^{n} \frac{w_i V(\mathbf{r}_i)}{x_i^2} \cdot (1 - U_i)^2 \tag{1a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \le m \quad \text{and} \quad \forall_i, \; U^{\min} \le U_i \le 1 \tag{1b}$$

where w_i is a weight assigned to event e_i, denoting its relative importance. By default, w_i is set to 1, but users can adjust it manually to specify the importance of events.

Online Solver. The above optimization problem is the same quadratic program as that presented in [14] to represent real-time elastic scheduling [12, 13]. The existing solver is tailored for single-core scenarios, whereas the problem under consideration involves multiple HPCs, which is conceptually equivalent to a multi-core scenario. Tintin-Scheduler adapts the solver by concatenating multiple HPC resources into a single virtual resource and scheduling events sequentially. This prevents any event from being scheduled concurrently on multiple counters, which would yield identical counts and not reflect the effective utilization assignment.

Formally, the "hyperperiod" of the schedule, denoted as H, represents the time after which the schedule repeats. For simplicity, it is normalized to $H = 1$. Events and counters are considered in order: event e_1 is assigned the interval $[0, U_1]$ on counter c_1. Events are placed sequentially on a counter until the total utilization on that counter would exceed 1, whereupon a portion of the event's utilization is assigned to the end of the hyperperiod interval on the current counter, and the remaining utilization is placed at the beginning of the interval on the next open counter. Since $U_i \le 1$, these intervals do not overlap in time. The schedule construction is illustrated in Figure 9, with the detailed procedure and an optimality sketch provided in Appendix A.

Event Groups. The Linux perf_event subsystem supports user-specified event groups, ensuring that even as events are multiplexed, those in the same group are always scheduled simultaneously. This is useful for identifying correlations among related events. Tintin-Scheduler extends this functionality, treating the group as a single schedulable entity. The total time allocated among all events in a group by the elastic scheduling policy is split evenly between them so all events are allocated to the same time slices across different counters.

Efficiency and Robustness Optimization. The solver retains the same runtime complexity as the original in [12], which is quadratic – $O(n^2)$ – in the number of events n. This is not negligible given the high scheduling frequency. Briefly, the solver iterates through all events to calculate their utilization. If any event produces a negative utilization, it must be set to zero, indicating that the event will not be scheduled. This requires backtracking and re-iterating through all the events. Consequently, the worst-case complexity is $O(n \times (n-1))$. With the insight that only low-uncertainty events produce negative utilization, the solver can prioritize high-uncertainty events, avoiding re-iterations. Leveraging this invariant, the solver first sorts all events by their uncertainty and then calculates utilization in a single pass. The overall complexity is quasilinear, only dominated by the sorting step, $O(n \log n) + O(n) = O(n \log n)$.

Furthermore, the calculated utilization could be extremely low in some cases. Besides the performance impact caused by closely triggered timer interrupts, such low utilization results in unstable estimations, as measured errors may be amplified when scaled to the full range. As such, Tintin-Scheduler imposes a lower bound $U^{\min}$ on the scheduling quantum (minimum schedulable utilization time slice), which is set to one-tenth of the hyperperiod in our implementation.

Event Starvation. A potential issue is that events with low uncertainty might not be scheduled, resulting in a persistently low uncertainty value. However, the uncertainty metric addresses this by increasing an event's uncertainty as the interval since its last scheduling grows, thereby ensuring that all events are eventually allocated on the HPC.

Implementation. Tintin-Scheduler is activated when Tintin-Monitor detects the end of a hyperperiod H, which is set by default to the CPU task scheduling interval. In addition to elastic scheduling, Tintin-Scheduler can be configured for simple round-robin multiplexing. A simpler priority-driven scheduling policy, *Uncertainty-First*, does not compute utilization, but instead prioritizes the events with the highest uncertainty at each hyperperiod. We note that support for PMU architectures that restrict some events from being assigned to certain HPCs is still limited, as existing algorithms to solve the restricted assignment problem with elastic scheduling are too complex for an efficient in-kernel implementation.

7 Tintin-Manager

Internally, Tintin-Manager is composed of two parts: a component for managing profiling scopes and HPC data, and an interface for handling user requests.

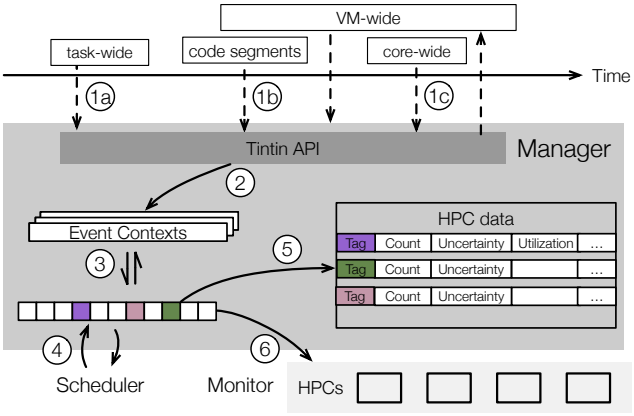


Figure 10: Tintin-Manager workflow. (1a) An *ePX* is created for a task, (1b) associated to code segments, or bound to a broader scope such as (1c) core-wide or VM-wide profiling. (2) Upon switching *ePX*s, Tintin-Manager updates outgoing event statistics and (3) fetches events for any incoming *ePX*s. It then (4) invokes Tintin-Scheduler to assign event priorities and time shares. It also (5) updates the HPC data if any events need to be scheduled out. Finally, (6) it places new events on the HPCs.

Event Profiling Context. To abstract heterogeneous profiling requirements, Tintin elevates profiling scope via a new first-class kernel object, the Event Profiling Context (*ePX*) abstraction. The *ePX* encapsulates the monitoring of all events associated with a given profiling scope, $ePX \sim \{e_i,..,e_j\}$. An *ePX* can be defined from the perspective of execution instances (e.g., a thread, a process, or all threads on a single core) and from the perspective of code segments (e.g., a function). Multiple contexts, ePX_m and ePX_n, may share common events e_k, but the HPC data (including counts and uncertainty) are maintained separately, with $e_k^m \rightarrow ePX_m$ and $e_k^n \rightarrow ePX_n$.

Contexts have been successfully used as first-class objects in prior work [40, 42, 51, 61], primarily for flexible privilege separation [17, 40, 51, 61] and time management [42]. Here, the *ePX* concept enriches hardware event profiling as follows.

Unified Management of *ePX*s. Figure 10 depicts the workflow and unified management approach of Tintin-Manager.

Flexible Definition of Profiling Scopes. Tintin-Manager translates the heterogeneous profiling requests from program segments and execution instances into a unified *ePX* format. Additionally, it supports *ePX* binding, enabling explicit scoping to arbitrary profiling scopes, such as multiple code segments or threads. The bound *ePX*s form a new *ePX*. This allows *ePX*s to be managed uniformly and collectively.

Correct Attribution of Event Counts. Tintin-Manager places calipers at an *ePX*'s entry and exit points to filter unrelated event counts, activating counting upon entry and deactivating it upon exit. To remain aware of *ePX* switches, Tintin-Manager listens for two types of signals. For *ePX*s associated with code regions, it utilizes instrumentation to insert specific syscalls at relevant code locations, triggering *ePX* switches.[3] For *ePX*s associated with execution instances, Tintin-Manager listens for CPU scheduling events to detect *ePX* switches.

Scope Conflict Resolution. To support overlapping profiling scopes, Tintin-Manager supports the simultaneous activation of multiple *ePX*s. Specifically, when more than one *ePX* is active, it groups all events and schedules them collectively, avoiding contention from overlapping events. An *ePX* may be assigned a weight, e.g., to avoid unfairness in resource allocation when it profiles numerous types of events. Tintin-Scheduler applies these weights as multipliers to the individual event weights within the *ePX*. Counts, variance, and uncertainty for each event are still maintained separately within each individual *ePX*. When an HPC is read, Tintin-Monitor attributes the count to all active *ePX*s monitoring that event, but their interpolated counts and uncertainty are updated individually. Upon *ePX* exit, Tintin-Manager removes the relevant events and reschedules.

On-the-fly Event Reconfiguration. Since *ePX*s are elevated to first-class objects, they can be shared and used by other components to reconfigure the events of interest dynamically within a profiling scope. This capability is crucial for applications that require adaptive profiling [35].

Programming API and Generality. In general, existing applications using perf_events may use Tintin instead without modification.

The perf_event Usage Model. Section §2.1 describes the Linux perf_event subsystem. In general, applications use it either via explicit instrumentation of syscalls, or by launching processes with the Perf command-line utility. In keeping with the Linux philosophy that "everything is a file," the driving syscall is *perf_event_open*, which takes several arguments indicating the profiling settings, and returns a file descriptor from which, among other data, HPC counts can be read.

Using Tintin Instead. A Linux kernel patched with Tintin switches between Tintin and the classical perf_event via an interface in procfs. When Tintin is selected, Tintin-Monitor and Tintin-Scheduler become active, but both the syscall API and Perf remain the same, allowing compatibility with existing applications. To access extended functionality, e.g., *ePX* creation and reported uncertainty values, Tintin-Manager provides the extended API outlined in Table 1. An *ePX* can be created for a given scope with a call to *tintin_create_context*. The API allows any *ePX* to be disabled on demand using *tintin_disable_context* and allows on-the-fly event modifica-

[3] A less intrusive alternative [62] is to leverage CPU debugging registers to insert interrupts at specific binary addresses without modifying the target executable; extending this functionality is left for future work.

tion with *tintin_add/remove_event*. Since an *ePX* identifier can be referenced globally, it can also be used to associate different *ePX*s using *tintin_associate_contexts* across different threads or processes, merging their events.

Table 1: Tintin-Manager API Functions

System calls	Descriptions
tintin_create_context(s)	Create an *ePX* at the specified scope *s*
tintin_enable_context(c)	Begin monitoring of the given *ePX c*
tintin_disable_context(c)	End monitoring of the given *ePX c*
tintin_add_event(e, c)	Begin monitoring event *e* in given *ePX c*
tintin_remove_event(e, c)	Stop monitoring event *e* in given *ePX c*
tintin_set_event_weight(e, w)	Set or modify event *e* scheduling weight *w*
tintin_associate_contexts(c_1, c_2)	Aggregate the counts of two *ePX*s
tintin_read_with_uncertainty(e)	Read both event count and uncertainty

Implementation. All APIs are implemented as system calls, wrapped in C. To reduce the effort required for instrumentation, we implemented a set of LLVM [36] compiler passes that automatically instrument the system calls during compilation. By utilizing different levels of passes (e.g., function-level, basic block-level), users can achieve various levels of granularity by adding a compilation flag. The current Tintin implementation defines profiling scopes in source code, but binary instrumentation [62] could extend this to binaries.

8 Evaluation

This section seeks to answer the following questions:

- How can Tintin be used in real-world applications? (§8.1)
- How effective is Tintin in improving event measurement accuracy? (§8.2)
- What is Tintin's overhead? (§8.3)
- How scalable is Tintin? (§8.4)

Experimental Setup. All experiments were conducted on a server with two Intel Xeon Gold 6130 (Skylake) CPUs and 32GB of RAM. Each CPU has 16 physical cores running at 2.10GHz with Hyper-Threading disabled. By default, the hyperperiod (scheduling cycle) in Tintin-Scheduler is set to 4 ms (250 Hz), aligning with the default scheduling interval of the Linux perf_event subsystem [65] on the tested machine. The scheduling quantum is set to $^1/_{10}$ the hyperperiod (0.4 ms). As part of our scalability analysis in §8.4, we explore the effects of these parameters with different values.

Ground Truth. Due to the non-determinism of modern CPUs, obtaining ground truth data outside of simulation is nearly impossible. Similarly to CounterMiner [41] and BayesPerf [2], we measure ground truth by pinning an event to an HPC, then reading its value on completion of a benchmark.

8.1 Case Studies

We use three case studies to demonstrate the effectiveness and usability of Tintin.

8.1.1 Dynamic Resource Provisioning with Pond

Target Application. Pond [37] is a resource orchestration subsystem developed by the Microsoft Azure team. In cloud environments, memory pooling may improve DRAM utilization and reduce cost, but using pooled instead of local memory might substantially increase the latency of some virtual machine workloads. To allocate the limited local memory appropriately, Pond predicts whether a VM workload's latency will be sensitive to the use of pooled memory.

Evaluation Methodology. Pond is a proprietary, closed-source system, so its authors do not provide the data or models used in [37]. Instead, we use Pond's open-source emulation layer [66] to reproduce its latency insensitive model [37, Figure 12] according to the paper's description. The predictive model is a random forest regressor implemented using Scikit-learn [52]. The model's inputs consist of 20 hardware events, sampled at 100 ms intervals via Intel's EMON tool [33]. To generate the training data, Pond runs the workload entirely on a local NUMA node, then again with memory from a remote node, reporting the execution time increase. We follow the experimental setup in [37], generating data using 12 workloads from SPEC CPU®2017 [11] and 30 from the GAP Benchmark Suite [3]. We randomly select 30 workloads for training and 12 for testing.

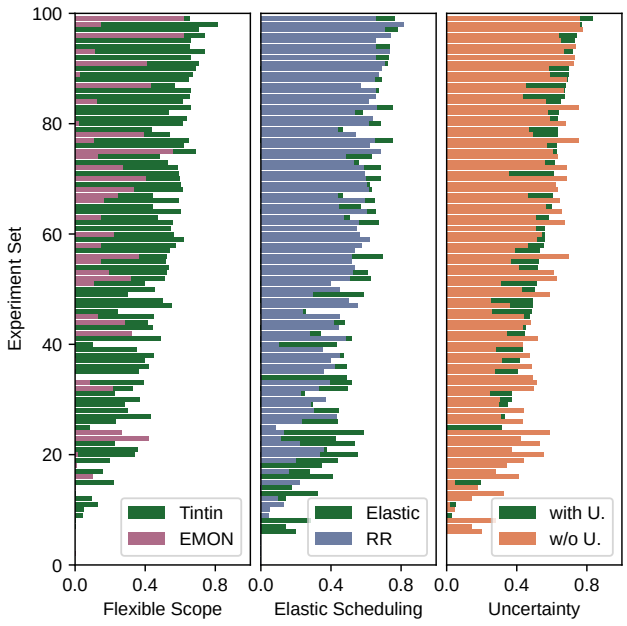


Figure 11: Results on Pond [37]. Each bar represents the score of the latency sensitivity prediction.

Results. A comparison of model prediction accuracy is shown in Figure 11; values closer to 1 indicate better performance.

Profiling Scopes. As Intel EMON only supports CPU-level event profiling, Pond's emulation layer pins VM threads to specified cores, then attributes all event counts from those cores to the VM. We modify Pond to instead use Tintin to associate an *ePX* with all VM threads. In the plot on the left, Tintin outperforms the baseline, measured by EMON [33], in 95 out of 100 experiments. On average, Pond's accuracy with Tintin is 0.51 higher than the baseline. This improvement is primarily due to Tintin's ability to attribute events directly via correctly defined profiling scopes.

Elastic Scheduling. To demonstrate the advantages of Tintin-Scheduler's elastic scheduling approach in reducing total uncertainty, we compare it to a round-robin scheduling strategy across another 100 experiment sets. The results are shown in the middle plot of Figure 11. Elastic scheduling outperformed the round-robin strategy in 64 out of 100 experiments, improving model prediction scores by an average of 0.15.

With Uncertainty. To demonstrate how user-space applications might use the uncertainty reported by Tintin, we modify our implementation of Pond's latency prediction model to include the uncertainty values reported by Tintin-Monitor as inputs. Results are illustrated in the right plot in Figure 11. With this additional information, scores are higher in 55 out of 100 cases compared to the model without uncertainty, and the average score increases by more than 0.02. The minor improvement is attributed to the fact that, in these tests, Tintin's targeted scope and scheduling policy succeed in minimizing uncertainty, so the small error values given to the model provide only incremental improvements.

Resolving Scope Conflicts. To evaluate Tintin's ability to resolve profiling scope conflicts, we modify the workloads managed by Pond to also require their own event monitoring, leading to potential conflicts with Pond's profiling scope. We instrument each workload to profile 16 events and then measure both the counting accuracy of the target workload and the prediction accuracy of Pond's latency model. The counting accuracy is measured by pinning a specific event (e.g., cache misses) to a dedicated hardware counter, which serves as the ground truth for comparison.

By default, Pond profiles events at the CPU core scope, and the Linux perf_event subsystem prioritizes these events over any other event bound to tasks. As a result, if other workloads attempt to profile events on the same core, their events may never be scheduled, leading to starvation. Tintin avoids this issue by collectively scheduling all events. Under this collective scheduling approach, measurement accuracy does not degrade significantly compared to separate runs that just profile each scope in isolation. From the workloads' perspective, the counting error increases slightly from 3.11% to 3.56%. From Pond's perspective, the prediction accuracy remains comparable, with an average score decrease of only 0.01 across 100 test cases. Moreover, the overhead of Tintin remains below 2.4% in both conflicting and non-conflicting scenarios, demonstrating that resolving conflicting scopes does not introduce significant overhead.

Cores	Metrics		% Slots
C0	FE	Frontend_Bound	39.0 <==
C0	BE	Backend_Bound	33.3
C0	RET	Retiring	19.5
C0	FE	Frontend_Bound.Fetch_Latency	28.7
C0	BE/Mem	Backend_Bound.Memory_Bound	23.8
C0	RET	Retiring.Heavy_Operations	12.5

(a) Results by Linux perf_event subsystem.

-	Metrics		% Slots
S1	BE	Backend_Bound	96.8 <==
S1	BE	Backend_Bound.Memory_Bound	90.9
S1	FE	Fetch_Latency.MS_Switches % Clocks_est	9.8
S1	BE/Mem	Backend_Bound.Memory_Bound	80.5 <==

(b) Results by Tintin.

Figure 12: Results on DMon.

8.1.2 Data Locality Problem Detection with DMon

A description of the target application, DMon [35], and its limitations has already been introduced in §3.

Evaluation Methodology. We use DMon's artifact [24] as a baseline for comparison and target the workload from its repository, which contains one data locality issue. Rather than using DMon's built-in time window, Tintin binds *ePX*s at the individual loop level via syscalls automatically instrumented at the beginning and end of each function in a compiler pass. This enables profiling different layers in Intel's Top-Down event hierarchy [73] every time a function is entered, eliminating the need to switch layers during function or loop execution. Since some functions may be invoked only once in a given run, a user may execute the target workload multiple times to ensure each layer is adequately profiled. By providing more targeted profiling scopes via its *ePX*s, Tintin ensures that events are correctly filtered for each function so as to correctly attribute the locality issue. In our case study, results are evaluated based on the accuracy with which they identify the code region experiencing Back-end Bound behavior.

Results. Visualization of DMon's output is shown in Figure 12. Among 10 experiment runs, DMon often results in an ambiguous mixture of Front-end and Back-end Bound instances, failing to identify data locality issues when using Linux perf_event. In Figure 12(a), the results indicate that most of the time is spent on the frontend. However, the program is actually Back-end Bound. Since DMon aggregates the counts over the program's execution, the events originating from Back-end Bound code segments are overshadowed by other normal code. In contrast, Tintin offers more precise profiling via loop-level scope. Notably, in Figure 12(b), the target function is identified as spending 96.8% of its time on the backend, directly indicating a data locality issue. Among

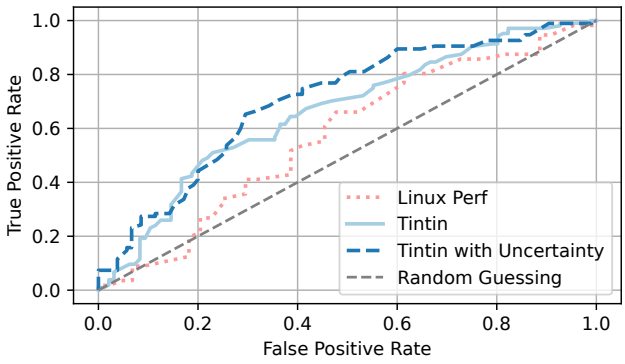


Figure 13: Accuracy of the random forest classifier in detecting the Linux rootkit.

10 repeated experiment runs, DMon failed in 9 out of 10 runs with perf_event. However, with Tintin, DMon reliably detects Back-end Bound issues; the reported backend time for the target consistently exceeds 91.1%.

8.1.3 Intrusion Detection System

HPC data has also been used for security purposes in the last two decades. Previous works [20, 34] leverage the measured HPC values to classify an unknown program as either benign or malicious. This case study investigates whether Tintin enhances the efficacy of intrusion detection.

Evaluation Methodology. We adopt the experimental setup from the seminal work of Demme et al. on detecting Linux rootkits with hardware event data [20]. As our malware, we use the open-source Linux rootkit Diamorphine [43], which is implemented as a loadable kernel module. To activate the rootkit's functionality, we execute common Linux commands, including ps, ls, whoami, printenv, and pwd, while monitoring their behavior. The rootkit is configured to hide processes, escalate user privileges to root, and conceal files. Since Demme et al. do not specify the hardware events used in their experiments, we select the 10 commonly used events predefined under PERF_TYPE_HARDWARE as described in [44].

We conducted three sets of experiments. The first set evaluates perf_event at the granularity of the target victim tasks. The second set evaluates Tintin's elastic scheduling. In this setup, we replaced the event scheduling logic with Tintin's scheduler but kept the rest unchanged. The third set extends the previous configuration by collecting uncertainty information at each measurement point. For each set, we collect 800 samples for training and 200 samples for testing; these are evenly split between execution scenarios with and without the rootkit. For the detection model, we adopt a random forest classifier, which is also used in [20, 77].

Results. We observed that using HPC data to detect malware within a single, fixed target task (e.g., ls) yields excellent classification results. However, this approach requires constructing a separate HPC-based malware signature for each individual task, which is impractical and limits generalizability. In our more realistic experimental setting, where the five target tasks vary, the detection performance is less compelling. Figure 13 presents the ROC curves for Linux perf_event, Tintin without uncertainty information, and Tintin with uncertainty information. The AUC (Area Under the Curve) for perf_event is 0.57. Tintin improves the AUC to 0.66 through its event multiplexing-aware scheduling. When uncertainty information is incorporated, the AUC further increases slightly to 0.70. Although the improvement is modest, these results indicate that Tintin provides benefits in this scenario.

Further Discussion. There are inherent limitations of relying solely on HPC data for intrusion detection because malicious software behavior may not always manifest in the specific microarchitectural events captured by HPCs [77]. Although our experiments showcase that Tintin raises the bar for attackers, sophisticated attacks can still evade detection.

8.2 Effectiveness of Tintin

For a more comprehensive evaluation, we use the SPEC CPU®2017 [11] and PARSEC 3.0 [9] benchmark suites to assess the accuracy of the event counts collected online by Tintin, comparing it to Linux perf_event and CounterMiner [41]. While CounterMiner is primarily intended for offline predictive modeling and analysis of events, it does attempt to reduce event count errors by smoothing outliers. For a reasonable runtime comparison, we implement its data cleaning phase in Linux perf_event. Another related infrastructure, BayesPerf, is not included in the comparison because it requires algebraic relations between events, which are absent from most event primitives in practice.

To evaluate the effectiveness of elastic scheduling, we compare it to Tintin-Scheduler's "Uncertainty-First" policy. In this configuration, events with higher uncertainties are prioritized, and the scheduled event occupies the HPC until the next hyperperiod. We select the 24 default predefined events in Linux perf_event to profile simultaneously [44]. To obtain ground truth, we pin one event to an HPC in each run. We repeat the experiments for the first 5 pre-defined events in PERF_TYPE_HARDWARE in [44]. The average errors are reported in Figure 14.

Event counts obtained with Linux perf_event were, on average, 9.01% off from the ground truth, with a maximum error of 53.27%. CounterMiner performed similarly with an average error of 8.80% and a maximum of 56.21%. In comparison, Tintin's errors remained well under 5% in most cases. Furthermore, while "Uncertainty-First" also reduces errors, it does not perform as well as elastic scheduling, achieving an average error of 6.51%, compared to 2.91%. This is due to two factors: first, its scheduling is heuristic rather than optimal; second, it cannot allocate finer-grained timeslices for

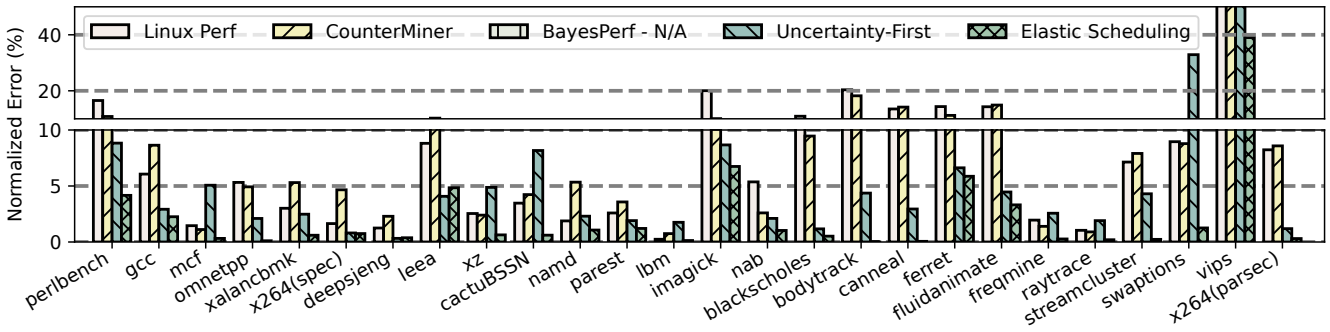


Figure 14: Measurement errors.

events, causing each event to occupy the HPC for an entire hyperperiod, which may lead to event starvation.

8.3 Runtime Overhead of Tintin

We also evaluate the runtime overhead of Tintin in comparison to Linux perf_event and BayesPerf [2]. We exclude CounterMiner [41] from these experiments, as it is a purely offline analysis tool. We instrument a CPU implementation of BayesPerf by modifying the Linux perf_event timer callback to run BayesPerf's Expectation Propagation algorithm [2, Alg. 1]. Figure 15 presents execution time means and standard deviations across 10 runs of each considered benchmark (normalized to the execution time without profiling). Without the dedicated hardware accelerator in [2], BayesPerf incurs excessive overhead, with up to 31.3% slowdown in the profiled benchmarks. This limits portability, as it cannot achieve acceptable runtime performance on standard hardware.

On the other hand, Tintin exhibits an average overhead of only 2.4%, only slightly higher than perf_event's 1.9%. This difference was found to be statistically insignificant. In the worst-case scenarios, we observed Tintin's overhead to reach up to 7.6% while perf_event's could be up to 12.7%. In some scenarios, Tintin achieves even better execution time performance than Linux perf_event due to its elastic scheduling method, which can trigger fewer interrupts.

8.4 Scalability Analysis

There are three factors impacting the accuracy and overhead of Tintin: scheduling hyperperiod, scheduling quantum, and number of monitored events. In this section, we examine how Tintin's performance scales with these factors. While the number of scopes might also impact scalability, these effects are reflected by the number of events.

Hyperperiod. We target the 505.mcf workload from the SPEC 2017 benchmark, varying the hyperperiod from 15 ms to 1 ms. The resulting changes in accuracy and overhead are recorded in Figure 16. We observe in Figure 16(a) that the accuracy improves with finer-grained hyperperiods, achieving an error rate of only 0.2% at 1 ms. In contrast, the error rate increases to 1.01% at a 15 ms hyperperiod. Meanwhile, perf_event exhibits significantly higher error rates across all hyperperiod settings, reaching up to 7.8% at 15 ms. Nonetheless, Figure 16(b) shows that while overhead increases with smaller intervals, it remains below 5%.

Scheduling Quantum. Since Linux perf_event schedules events at the hyperperiod, it does not support tuning the scheduling quantum. Consequently, we exclude experiments involving it. The accuracy and overhead results are presented in Figure 17(a). Accuracy stabilizes at 0.5 ms, with an error rate of 0.6%. Reducing the scheduling quantum further does not significantly enhance accuracy but noticeably increases overhead. For instance, at a quantum of 0.05 ms, the overhead surpasses 7.5%.

Number of Events. We evaluate the scalability of elastic scheduling in handling varying numbers of events, as shown in Figure 17(b). The error remains below 5.4% with 512 event types, but when the number reaches 1024, Tintin can render the kernel unresponsive. This issue occurs when TintinScheduler is triggered during a CPU context switch. To generate a new schedule, it sorts all events by uncertainty, which takes too long with 1024 events. As a result, it may not finish before the next jiffy, causing the CPU scheduler to miss the deadline and be deemed unresponsive. In practice, the number of profiled events is usually less than 256, so Tintin remains usable. In contrast, the errors in perf_event increase more significantly with the number of events, exceeding 20% when monitoring 256 events.

9 Related Work

Measurement Error. Most existing work that avoids multiplexing does so by running a target application multiple times, each time monitoring a different subset of events. Results are aligned into a single trace [31, 41, 48] using techniques such as dynamic time warping [7] or uniform scaling [47]. However, these offline techniques are unsuitable for online monitoring, which is required in many applications such as resource orchestration [37, 54] and performance interference

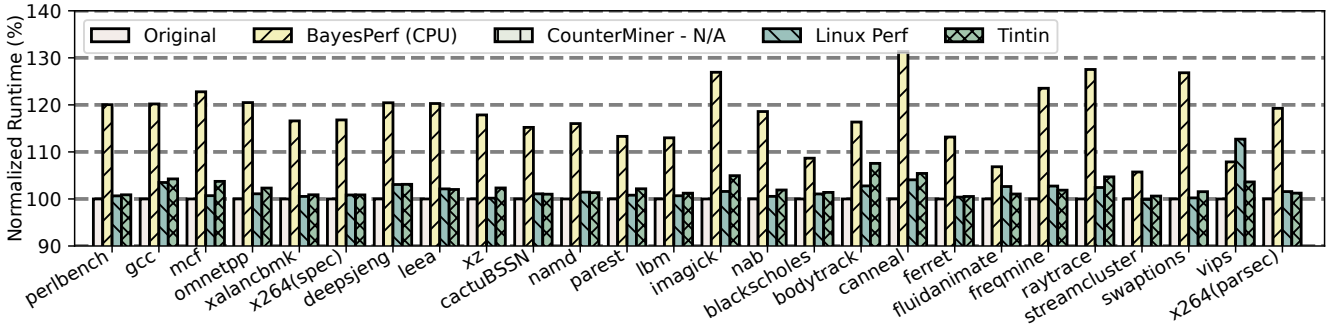
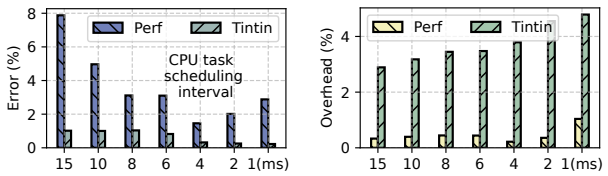
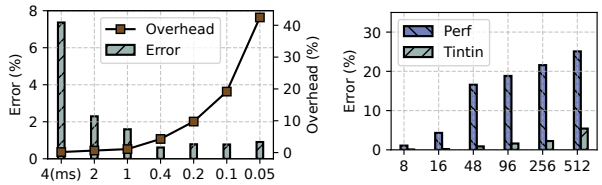


Figure 15: Runtime overhead.



(a) Accuracy. (b) Overhead.

Figure 16: Scalability analysis of event scheduling hyperperiod.



(a) Scheduling quantum. (b) Number of events.

Figure 17: Scalability analysis scheduling quantum and event count.

mitigation [25, 54, 72]. Additionally, merging runs may fail to identify some performance anomalies, as performance can vary widely across runs, with anomalies only appearing in some.

BayesPerf [2] schedules events online, constructing event groups to guarantee statistical dependencies between events in adjacent groups, thus inferring the counts of unscheduled events. However, it has three main drawbacks. First, it depends on algebraic relationships among events, which are unavailable for most event primitives on platforms like Intel [18], IBM [30], and ARM. Second, it incurs prohibitive overhead, necessitating customized hardware for acceleration. Third, its scheduling algorithm is designed offline, preventing adaptation for dynamic workloads. With elastic scheduling, Tintin overcomes these limitations and achieves accuracy improvements comparable to those reported in prior work [2].

Attribution Error. Existing work on attribution falls into two main categories. The first focuses on precise event filtering for target profiling scopes, achieved either through efficient instrumentation during compilation [1, 62] or by using techniques for post-mortem analysis [64]. The second addresses conflicts within overlapping profiling scopes. For example, Metis [71] virtualizes HPCs using time division, allowing multiple users in different virtual machines to monitor various CPU events simultaneously. Tintin unifies these functionalities across various use cases through the *ePX* and system support.

Another line of research focuses on architectural misattribution caused by skid effects, which is discussed in §2.2. Addressing this issue often requires hardware redesign [28, 29]. Software-based methods, such as padding with dummy instructions, are also an alternative [74] but incur significant overhead. These are out of scope for this paper.

10 Conclusion

This paper introduces Tintin, a hardware event profiling infrastructure designed to address event multiplexing errors and the inflexibility of profiling scope definitions. Designed with a three-component structure, each component of Tintin individually mitigates these tensions to some degree, while also collaboratively working together for enhanced overall performance. The evaluation results showcase that Tintin provides flexibility in profiling and improves profiling accuracy with minimal overhead.

Acknowledgment

We thank our shepherd, Pedro Fonseca, and the anonymous reviewers for their valuable feedback. We also thank Tobias Pristupin and Lien Zhu for their contributions to the implementation. This work was supported by the NSF (CNS-2154930, CNS-2229427, CNS-2141256, CNS-2403758, CNS-2229290), the ARO (W911NF-24-1-0155), the ONR (N00014-24-1-2663), a WashU OVCR seed grant, and Intel.

References

[1] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[2] Subho S. Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Bayesperf: Minimizing performance monitoring errors using bayesian statistics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 832–844, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[4] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. Profiling dataflow systems on multiple abstraction levels. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 474–489, 2021.

[5] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 177–191, USA, 2010. USENIX Association.

[6] Emery D Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with {SCALENE}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 51–64, 2023.

[7] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, page 359–370, Seattle, WA, 1994. AAAI Press.

[8] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 103–116, USA, 2008. USENIX Association.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, aug 2000.

[11] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[12] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, page 286, USA, 1998. IEEE Computer Society.

[13] Giorgio C. Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

[14] Thidapat Chantem, Xiaobo Sharon Hu, and M. D. Lemmon. Generalized elastic scheduling. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, page 236–245, USA, 2006. IEEE Computer Society.

[15] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 42–52, New York, NY, USA, 2010. Association for Computing Machinery.

[17] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[18] Intel Corp. Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm, 2016. Accessed 2019-03-05.

[19] Sanjeev Das. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Symposium on Security and Privacy (SP)*, pages 20–38, Oakland,CA,USA, 2019. IEEE, IEEE.

[20] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 559–570, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. Reliable and efficient performance monitoring in linux. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, Salt Lake City, Utah, 2016. IEEE Press.

[22] Manfred Drosg. *Dealing with uncertainties*. Springer, 2007.

[23] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[24] EfesLab. Dmon-ae. https://github.com/efeslab/DMon-AE, 2023. Accessed: 2023-11-26.

[25] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[26] Crispin W. Gardiner. *Stochastic Methods: A Handbook for the Natural and Social Sciences*. Springer, 4th edition, 2009.

[27] Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. Dna: Dynamic resource allocation for soft real-time multicore systems. In *27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 196–209, Nashville, TN, USA, 2021. IEEE.

[28] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. Tip: Time-proportional instruction profiling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 15–27, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. Tea: Time-proportional event analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Brian Hall, Peter Bergner, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Mauricio Oliveira, Bill Schmidt, Will Schmidt, et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017.

[31] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 281–296, New York, NY, USA, 2005. Association for Computing Machinery.

[32] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[33] Intel Corporation. Emon user's guide. https://www.intel.com/content/www/us/en/content-details/686077/emon-user-s-guide.html, 2023. Accessed: 2023-12-01.

[34] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[35] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 163–181, Virtual, July 2021. USENIX Association.

[36] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[37] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Arm Limited. Arm cortex-a53 technical reference manual r0p4. https://developer.arm.com/documentation/ddi0500/j/Performance-Monitor-Unit/Events?lang=en. Accessed 2023-11-16.

[39] Arm Limited. Arm cortex-a78 technical reference manual. https://developer.arm.com/documentation/101430/0102/Debug-descriptions/Performance-Monitoring-Unit/PMU-events. Accessed 2023-11-16.

[40] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. {Light-Weight} contexts: An {OS} abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, 2016.

[41] Yirong Lv, Bin Sun, Qinyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. Counterminer: Mining big performance data from hardware counters. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 613–626, Fukuoka, Japan, 2018. IEEE Press.

[42] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[43] m0nad. Diamorphine - lkm rootkit for linux kernels. https://github.com/m0nad/Diamorphine, 2023. Accessed: 2025-04-06.

[44] man7.org Linux Man-pages project. *perf_event_open(2) - Linux Manual Page*. Accessed: 2024-11-02.

[45] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 409–425, USA, 2018. USENIX Association.

[46] John M May. Mpx: Software for multiplexing hardware performance counters in multithreaded programs. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 8–pp. IEEE, 2001.

[47] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. Time interpolation: So many metrics, so few registers. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, page 286–300, USA, 2007. IEEE Computer Society.

[48] Richard Neill, Andi Drebes, and Antoniu Pop. Fuse: Accurate multiplexing of hardware performance counters across executions. *ACM Trans. Archit. Code Optim.*, 14(4), dec 2017.

[49] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 377–389, USA, 2017. USENIX Association.

[50] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 97–108, New York, NY, USA, 2009. Association for Computing Machinery.

[51] Gabriel Parmer. The case for thread migration: Predictable ipc in a customizable and reliable os. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2010)*, page 91, 2010.

[52] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[53] perf event. https://www.brendangregg.com/linuxperf.html. Accessed: 2022-11-13.

[54] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Firm: An intelligent fine-grained resource management framework for slo-oriented microservices. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[55] James Reinders. *VTune performance analyzer essentials*, volume 9. Intel Press, Santa Clara, 2005.

[56] Jerome H. Saltzer. Protection and control of information sharing in multics. In *Proceedings of the Fourth ACM Symposium on Operating System Principles*, SOSP '73, page 119, New York, NY, USA, 1973. Association for Computing Machinery.

[57] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[58] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. *ACM SIGARCH Computer Architecture News*, 41(1):65–76, 2013.

[59] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGPLAN Not.*, 37(10):45–57, oct 2002.

[60] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[61] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222, 2010.

[62] Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. Pinpointing performance inefficiencies via lightweight variance profiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Marion Sudvarg, Chris Gill, and Sanjoy Baruah. Linear-time admission control for elastic scheduling. *Real-Time Systems*, 57(4):485–490, Oct 2021.

[64] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 441–452, New York, NY, USA, 2009. Association for Computing Machinery.

[65] Linus Torvalds and Contributors. Linux kernel perf events directory. https://github.com/torvalds/linux/tree/master/kernel/events, 2024. Accessed: 2024-04-27.

[66] Vtess. Pond emulation. https://github.com/vtess/Pond. Accessed: December 5, 2023.

[67] Vincent M Weaver and Sally A McKee. Can hardware performance counters be trusted? In *2008 IEEE International Symposium on Workload Characterization*, pages 141–150, Seattle, USA, 2008. IEEE.

[68] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224, Austin, TX, 2013. IEEE.

[69] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[70] Perf Wiki. Tutorial: Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial, 2023. Accessed on: 2023-11-16.

[71] Xia Xie, Haiou Jiang, Hai Jin, Wenzhi Cao, Pingpeng Yuan, and Laurence Tianruo Yang. Metis: a profiling toolkit based on the virtualization of hardware performance counters. *Human-centric Computing and Information Sciences*, 2:1–15, 2012.

[72] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. Dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[73] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[74] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 98–105, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So many performance events, so little time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[76] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. Happy: Hyperthread-aware power profiling dynamically. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 211–218, USA, 2014. USENIX Association.

[77] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 457–468, New York, NY, USA, 2018. Association for Computing Machinery.

Algorithm 1: Elastic_Scheduling

Input: Lists $\{U_i^{\min}\}, \{U_i^{\max}\}, \{E_i\}$ sorted in non-decreasing order of $\frac{U_i^{\max}-U_i^{\min}}{E_i}$ and a desired utilization U_D

Output: The list of assigned U_i values

$U_{\text{SUM}} \leftarrow 0$; $E_{\text{SUM}} \leftarrow 0$; $\Delta \leftarrow 0$
forall $\tau_i \in \Gamma$ **do**
  $U_{\text{SUM}} = U_{\text{SUM}} + U_i^{\max}$
  $E_{\text{SUM}} = E_{\text{SUM}} + E_i$
end
forall $\tau_i \in \Gamma$ **do**
  if $\left(U_i^{\max} - \frac{U_{\text{SUM}}-(U_D-\Delta)}{E_{\text{SUM}}} \times E_i \le U_i^{\min}\right)$ **then**
    $U_i \leftarrow U_i^{\min}$
    $\Delta \leftarrow \Delta + U_i^{\min}$
    if $(\Delta > U_D)$ **then return** INFEASIBLE
    $U_{\text{SUM}} \leftarrow U_{\text{SUM}} - U_i^{\max}$
    $E_{\text{SUM}} = E_{\text{SUM}} - E_i$
  else
    $U_i \leftarrow U_i^{\max} - \frac{U_{\text{SUM}}-(U_D-\Delta)}{E_{\text{SUM}}} \times E_i$
  end
end
return FEASIBLE

A Solving the Scheduling Problem

The scheduling problem in Section 6 assigns a utilization to each event of interest to minimize total expected normalized error, with additional weights assignable by the user. This is expressed as the following constrained optimization problem:

$$\min_{U_i} \quad \sum_{i=1}^{n} \frac{w_i V(\mathbf{r}_i)}{x_i^2} \cdot (1-U_i)^2 \tag{2a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \le m \tag{2b}$$

$$\forall_i \quad U^{\min} \le U_i \le 1 \tag{2c}$$

where U_i is the utilization assigned to event e_i, w_i is a user-defined weight, $V(\mathbf{r}_i)$ is the observed variance, and x_i is the estimated count. m denotes the number of available HPCs. This reduces in polynomial time to the formulation in [14] of elastic scheduling as a constrained optimization problem,

$$\min_{U_i} \quad \sum_{i=1}^{n} \frac{1}{E_i}(U_i^{\max}-U_i)^2 \tag{3a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \le U_D \tag{3b}$$

$$\forall_i, \ U_i^{\min} \le U_i \le U_i^{\max}, \tag{3c}$$

by setting $U_i^{\max} = 1$, $E_i = \frac{x_i^2}{w_i V(\mathbf{r}_i)}$, and $U_D = m$.

A.1 Optimal Solution Algorithm

From [63, Algorithm 1], this can be solved in quasilinear time by the procedure listed in Algorithm 1.

A.2 Constructing a Schedule

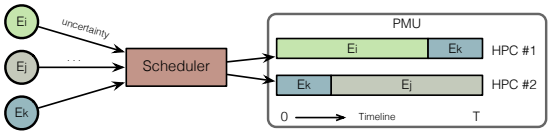


Figure 18: Constructing a schedule of 3 events on 2 HPCs from assigned utilizations.

Although the adopted elastic scheduling problem targets uniprocessor scheduling, we now show that our problem admits a valid schedule for events on *multiple* HPCs. We denote by H the hyperperiod of the schedule, after which it repeats itself; for simplicity of explication we normalize to $H = 1$. Events and counters are considered in order: event e_1 is assigned the interval $[0, U_1]$ on counter c_1. Events are placed sequentially on a counter until total counter utilization would exceed 1, whereupon an event's utilization is split between the end of the hyperperiod interval on the current counter and the beginning of the interval on the next open counter. This continues until all events have been scheduled. This method guarantees that no event is scheduled concurrently on two counters at the same time: if an event is assigned to two counters, the assignments are respectively at the end and beginning of the hyperperiod. Since $U_i \le 1$, these intervals do not overlap, as illustrated in Figure 18.





Building Bridges: Safe Interactions with Foreign Languages through Omniglot

Leon Schuermann[†], Jack Toubes[†], Tyler Potyondy[‡], Pat Pannuto[‡], Mae Milano[†], Amit Levy[†]

†*Princeton University,* ‡*University of California, San Diego*

Abstract

Memory- and type-safe languages promise to eliminate entire classes of systems vulnerabilities by construction. In practice, though, even clean-slate systems often need to incorporate libraries written in other languages with fewer safety guarantees. Because these interactions threaten the soundness of safe languages, they can reintroduce the exact vulnerabilities that safe languages prevent in the first place.

This paper presents Omniglot: the first framework to efficiently uphold safety and soundness of Rust in the presence of unmodified and untrusted foreign libraries. Omniglot facilitates interactions with foreign code by integrating with a memory isolation primitive and validation infrastructure, and avoids expensive operations such as copying or serialization.

We implement Omniglot for two systems: we use it to integrate kernel components in a highly-constrained embedded operating system kernel, as well as to interface with conventional Linux userspace libraries. Omniglot performs comparably to approaches that deliver weaker guarantees and significantly better than those with similar safety guarantees.

1 Introduction

Systems built using type-safe and memory-safe programming languages are safer, more reliable, and more secure than their unsafe counterparts. Languages like Swift, Go, and Rust promise to eliminate entire classes of bugs by construction. While the advantages of memory and type safety have been understood for decades [5, 15], recent advances in making safe languages more practical, along with encouragement from government agencies [48], have fostered their adoption in real, practical systems by both researchers [6, 21, 30, 31] and practitioners [7, 9, 47].

At the same time, real-world systems often require incorporating components written in several languages. As legacy and new systems evolve to use newer, safer languages [7, 9], both old and new languages must co-exist, potentially for years. Components such as filesystems, networking stacks, and device drivers have been fine-tuned over decades by scores of engineers. Pragmatic development does not immediately discard such thoroughly tested code but incrementally replaces it with safe languages when new features or complex fixes would otherwise be required.

Unfortunately, integrating foreign libraries into type-safe programs can re-introduce the exact vulnerabilities safe languages eliminate. When foreign code is invoked, it runs in the same address space and with the same privileges as the host language. A bug in a foreign library, such as the infamous OpenSSL Heartbleed vulnerability [39], can arbitrarily violate the safety of the host language by, for example, accessing memory that the host language assumed was private.

Recent contributions propose isolating foreign code in separate protection domains [4, 16, 29, 35]. Unfortunately, memory isolation *alone* is insufficient to maintain safety. Differences in semantics across languages mean that even interactions with internally correct foreign libraries—ones that operate only on sandboxed memory—may violate safety in subtle ways. For example, two languages may differ in their restrictions on pointer aliasing and corresponding types may have subtly different memory layouts or permissible values. Manually enforcing these invariants and translating types between different languages is error prone and endangers security and reliability [25]. Instead, we need a safe Foreign Function Interface (FFI) that maintains memory safety as well as all other language invariants.

We present Omniglot, the first framework to efficiently maintain all of Rust's safety-critical language invariants in the presence of foreign code. Rather than expose foreign functions directly to the host program, Omniglot invokes foreign functions through a memory sandbox and wraps returned values and shared memory in special types that ensure host code only uses foreign objects once they are validated to adhere to the host language's requirements. By designing a safe FFI using these types, in contrast to prior work, Omniglot maintains both memory and type safety of the host program. Importantly, Omniglot does so while avoiding expensive operations such as copying or (de)serialization across the FFI, which retains the efficiency of raw, unchecked FFI transitions.

Our implementation of Omniglot for the Rust programming language works with arbitrary foreign libraries and with any memory isolation mechanism that meets certain common criteria (Section 4.2). We implement Omniglot for use in a Rust operating system kernel using RISC-V's Physical Memory Protection (PMP) and in Linux userspace applications using x86 Memory Protection Keys (MPK).

Finally, we modify the widely used FFI binding generator for Rust, `rust-bindgen`, to generate foreign library bindings for Omniglot. Our evaluation shows that this allows developers to effectively use Omniglot in both kernels and userspace programs for a variety of practical libraries, such as cryptography, compression, image decoding, filesystem and TCP/IP networking. Omniglot has negligible overhead compared to memory isolation alone, practical overhead compared to an unchecked FFI, and performs significantly better than related work that copies and (de)serializes shared data.

2 Background & Motivation

Many of the safety advances in modern systems programming languages are rooted in their strict enforcement of memory and type safety. This enforcement allows these languages to leave out runtime-checks otherwise required in unsafe languages. For example, whereas a C string (`const char *`) may contain arbitrary 8 bit values, Rust strings (`str`) are guaranteed at compile-time to include only UTF-8 codepoints [40]. However, safe languages' reliance on such properties makes their enforcement critical. A single, subtle violation of memory safety, type safety, or other important invariants can lead to arbitrary undefined behavior and have significant implications for security and reliability.

Unfortunately, current tools used to facilitate interactions between languages do not capture many of these invariants. This deficiency means that all interactions with foreign code must be presumed *unsafe*. Instead, these tools place the proof burden of safety onto the developer who has to verify not only that foreign code is correct, but also that any interactions with this code maintain both the host and guest (foreign) language's invariants.

In this section, we show how current cross-language bindings insufficiently capture language invariants and illustrate how this can violate Rust's safety properties.

2.1 Cross-Language Interactions

Code written in different programming languages can interact through a Foreign Function Interface (FFI). This interface allows one language to invoke foreign language functions by emulating the Application Binary Interface (ABI) and calling conventions of the foreign language. As it is infeasible to support the calling conventions of every other language, many programming languages feature support for the C ABI as a universal, intermediate interface to facilitate interactions between high-level languages. As we will see throughout this section, the C ABI encodes virtually no assumptions about a language's properties such as its memory model, which values are valid for a given type, and synchronization behavior. While this flexibility makes the C ABI useful to connect languages that differ in these properties, it also places a heavy burden on developers to maintain correctness across FFIs.

crypt.h:

```
bool aes_encrypt(
  const uint8_t *key, uint8_t *src,
  uint8_t **const dst, size_t len, bool in_place);
```

crypt.rs:

```
extern "C" {
  // Implicitly marked as `unsafe`:
  pub fn aes_encrypt(
    key: *const u8, src: *mut u8,
    dst: *mut *const u8, len: usize,
    in_place: bool) -> bool;
}
```

Listing 1: Simplified Rust bindings for a C header file `crypt.h` as generated by `rust-bindgen`. The Rust binding is wrapped in an **`extern`** `"C"` block and uses raw pointers instead of Rust references.

Rust developers can use the `rust-bindgen`[1] tool to automatically generate a set of Rust bindings from a C header file. Listing 1 shows a C function signature and its corresponding Rust binding function as generated by `rust-bindgen`. The **`extern`** `"C"` tells the Rust compiler to invoke symbols using the C ABI. External function definitions are automatically tagged as **`unsafe`**—this attribute indicates that the Rust compiler cannot reason about the safety of this function and instead shifts the burden of proof to the developer. Consequently, developers must surround the call-sites of such functions with an **`unsafe`** block promising to the compiler that they verified the safety implications of the function call. For instance, Listing 2 shows an attempt at writing a safe wrapper around the foreign function of Listing 1.

Throughout this section we use the function `aes_encrypt` in Listing 1 as a running example. This function takes an encryption key and a `src` buffer with unencrypted data. Depending on the value of `in_place`, it can either encrypt data in-place, overwriting the contents of `src`, or by allocating a new buffer. For both modes, `dst` will be set to a pointer to the encrypted buffer.

2.2 Memory Safety

Guest (foreign) code has unrestricted access to the host code's memory, even when not explicitly provided to a foreign function. This is a problem. In Listing 2, if `Vec src`'s internal length attribute were stored one memory word before the

[1] https://github.com/rust-lang/rust-bindgen

```rust
pub enum Message<'a> {
  Encrypted(bool, &'a [u8]),
  Unencrypted(CString),
}

pub fn enc(src: &mut Vec<u8>) -> Message<'_> {
  let mut dst: *const u8 = ptr::null();
  unsafe {
    // Calling aes_encrypt from Listing 1
    let res = aes_encrypt(KEY, src.as_mut_ptr(),
      &mut dst, src.len(), IN_PLACE);
    Message::Encrypted(res,
      slice::from_raw_parts(dst, src.len()))
  }
}
```

Listing 2: A Rust wrapper around the C library function `aes_encrypt` from Listing 1, which returns a **`bool`** value to indicate whether the encrypt-operation succeeded. The Rust program exposes a function `enc` that takes a byte-array and uses the C library function to encrypt its contents. It returns the `Encrypted` variant of the **`enum Message`** type, a tagged union. This variant includes a buffer constructed from the pointer written to `dst` and the encryption-operation's result. Can you spot the safety violations?

buffer's contents, then a simple off-by-one error in the C function could overwrite this value (at address `src - 1`). This can then cause Rust code to believe that this array is larger than its actual underlying allocation.

In Rust, `Vec` is an instance of a safe abstraction. It provides an API through which the programmer is unable to violate Rust's safety invariants. When changing the internal length parameter, the C function prevents this API from living up to its promise, as the `Vec` may overflow its allocation.

Rust prevents such unsoundness by only providing functions with references to valid `Vec` instances. Unlike the raw pointers required by the unchecked C ABI, references do not allow a developer to perform arbitrary reads and writes to their memory. Instead, references are well-typed, and all accesses to their memory must be made through the type's API.

To maintain memory safety and thus *soundness*—crucial to security and integrity—interactions with foreign code must uphold an equivalent set of restrictions. In practice, for the example in Listings 1 and 2 this means that the `aes_encrypt` function, being passed a pointer to the `src` vector, must only read and modify the user-accessible part of the `Vec` data structure. The caller must, in turn, ensure that the vector outlives this function call.

2.3 Aliasing and Mutability

Apart from basic memory safety, programming languages often have additional, high-level invariants they require for correctness and optimizations. A prominent example of this

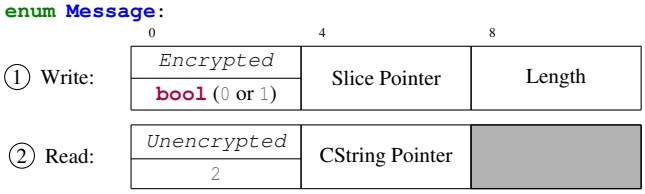


Figure 1: Memory layout of the **`enum Message`** type. Rust encodes the **`bool`** value of the `Message::Encrypted` variant into the same memory location as the enumeration's active variant (called *niche filling*). When ① writing an `Encrypted` value with a slice and an invalid **`bool`** value (*e.g.*, 2), the value may be ② misinterpreted as a different variant, such as `Unencrypted` and a `CString`.

are Rust's rules around aliasing and mutability. McCormack et al. explain these rules in great detail [25]; for brevity we illustrate them through the example of Listing 1.

Such an idiomatic C function interface can easily cause issues when paired with Rust's restriction on *aliasing XOR mutability*. This rule governs that at any time a value may either have multiple immutable *shared* references, or a single mutable *unique* reference, and never both. The `enc` wrapper of Listing 2 violates this rule: it invokes `aes_encrypt` with a `src` pointer obtained from a mutable reference to the data contained in the `src` vector (line 10). To pass on the encrypted data as part of the `Message` enum, it creates an immutable reference to this buffer (a *slice*) from the `*dst` pointer (line 13). However, when `aes_encrypt` is set to perform an *in-place* operation, `*dst` will be pointed to the exact same memory as `src`. Thus, creating an immutable reference from `*dst` is unsound while the original mutable reference for `src` still exists. Therefore, developers must reason about the provenance of pointers as they flow through both host and guest (foreign) code—a task that is difficult in practice, infeasible without source code, and where getting it wrong is commonplace [25] and leads to critical security vulnerabilities [1].

2.4 Type Safety and Valid Values

Finally, safe languages rely on type safety to maintain safety-invariants at runtime. In Rust, examples of this are the UTF-8 validity requirement for strings mentioned previously or the `Vec` abstraction shown in Section 2.2. In addition to these high-level invariants, Rust places restrictions onto the valid values of primitives and algebraic data types. For example, the **`bool`** primitive is a type occupying one byte of memory, whose only valid values are `0` or `1`. Treating memory containing any other value as a **`bool`** is thus unsound.

The implications of this are not solely of theoretical interest, though, as is apparent by surprising runtime behavior of Listing 2 when this invariant is violated: it may crash with

a segmentation fault![2] This is because Rust performs an optimization known as *niche filling*. Since Rust trusts that the `bool` type will only ever contain `0` or `1`, it encodes the value of this type directly into the discriminant of the `enum Message` type, and represents the `Unencrypted` variant with a discriminant[3] of value `2`. Figure 1 illustrates the memory layout of this type. However, if we were to violate the aforementioned validity restriction of `bool` we can construct a `Message::Encrypted` instance that, when read back, will be misinterpreted as an `Unencrypted` variant instead. This is not farfetched. For example, in many C implementations, a `bool` is `typedef`ed as an `int`, and hence may contain values outside of `0` or `1`. Because Rust assumes the returned `bool` is valid (`0` or `1`), it simply copies its memory into the `enum Message`'s discriminant and performs no further validations. As such, the memory occupied by the slice (`&'a [u8]`) member in Figure 1 will erroneously be interpreted as an element of type `CString`, leading to undefined behavior and a soundness violation in theory, and a use-after-free or out-of-bounds access in practice.

3 The Omniglot Approach

Interactions with foreign languages through Rust's existing *unsafe* FFI can trivially reintroduce the exact security and reliability vulnerabilities that safe Rust eliminates. Through this FFI, foreign code has unrestricted access to host memory. Moreover, even with coarse grained memory access restrictions, a foreign library could still introduce vulnerabilities due to subtle differences in language semantics.

Omniglot provides a *safe* interface for Rust programs to interact with foreign libraries. This is a high bar. Safety in Rust relies on *soundness*, i.e., interactions with a foreign library must not introduce any undefined behavior [44]. This includes[4] an absence of data races, dangling or misaligned accesses, aliasing violations, producing invalid values, etc.

Unfortunately, reasoning about whether a foreign library violates any of Rust's soundness conditions is hard. One conventionally achieves this by modeling the entire program and ensuring that the host code, foreign code, and their composition is sound. While the Rust compiler validates soundness of safe Rust code, it cannot do so for interactions with foreign code. Despite this difficulty, maintaining soundness is vital[5] to leverage the safety guarantees of type-safe languages.

[2] In practice, compiler optimizations happen to often avoid faults in simple cases like `bool` and other `enum` types with a 1 bit discriminant. For brevity, we elide an example that faults with a recent Rust compiler toolchain.

[3] The actual value chosen by the Rust compiler is not specified; at the time of writing it happens to use `2` for 32 bit platforms.

[4] Rust defines undefined behavior through its *Rust Reference* [41], *Unsafe Code Guidelines Reference* [44], and *Rustonomicon* [42]. In the absence of a formal model of Rust's semantics there is no exhaustive list of defined or undefined behavior; as such we design Omniglot against the currently known invariants.

[5] The Rust Unsafe Code Guidelines Reference states more bluntly that in the presence of undefined behavior, "the program produced by the compiler is essentially garbage." [44]

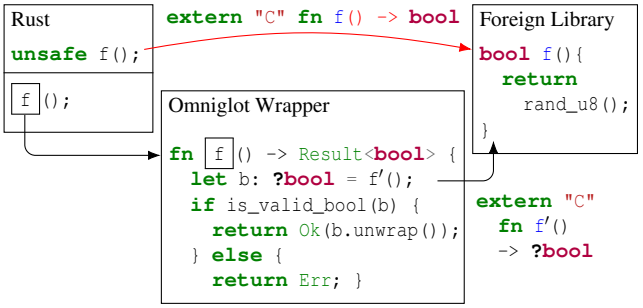


Figure 2: Omniglot interposes on interactions between Rust and foreign code. In this example, Omniglot models a foreign function through a *weaker* function binding `fn f'()`, and restores the original function's return type through a runtime check in the wrapper `fn f()`.

Omniglot does not model the entire program, and instead takes a different approach. We argue that it is sufficient to reason about an execution of the foreign library, as opposed to proving the soundness of the composition of host and foreign code for all such executions. Figure 2 demonstrates this approach: `f()` is a foreign function that would violate Rust's soundness through a naïve FFI binding. When a Rust program calls this function, the compiler *assumes* certain invariants hold across the foreign function's execution (we enumerate this set of invariants below). In this example, `f()` violates one such invariant by returning a potentially invalid value for the Rust `bool` type (*e.g.*, the number `2`).

Nonetheless, Omniglot can provide a safe encapsulation of this foreign function, `f`. Recall that Rust's invariants for a given value depend on its ascribed type. The problem with the example of Figure 2 arises from the fact that `f()` is said to return a value of the Rust type `bool`. Omniglot instead produces a foreign function binding `f′`. This binding returns a different type, `?bool`, which is not subject to those same restrictions. As such, calling `f′` does not endanger soundness.

While the foreign function binding `f′` is sound, it is not very useful. Because `?bool` is not subject to the same invariants as Rust's regular `bool` type, we also give up on the guarantees that follow from these invariants. Omniglot re-establishes these invariants through a set of runtime checks. In the above example, Omniglot tests whether the returned value is indeed a valid `bool` and, if so, casts it to this more useful type. This allows Omniglot to safely expose an encapsulated function `f` with the same interface as its unsound counterpart, `f`.

This example shows how we reason about type safety invariants to develop Omniglot's safe foreign function interface. In addition to type safety, there are also memory safety, aliasing and mutability, and concurrency-related invariants. Over

the remainder of section, we describe these remaining invariants, and show how we can uphold them in a similar way.

Memory Safety. We adopt Dhurjati et al.'s definition of memory safety: a software entity is memory-safe if it only references memory allocated by or for that entity [10]. As this is ultimately a language-specific property, strictly enforcing it across an FFI is difficult. For instance, moving an object—a common operation in Rust—has no equivalent counterpart in a language like C with no defined ownership model.

However, Omniglot only needs to retain the *host* language's memory safety invariants, a more tractable problem. By dividing each language's allocations into a distinct set of memory regions, where each side is restricted from accessing the other's allocations, we ensure that any memory safety issues within foreign code cannot impact the surrounding Rust program. Memory isolation techniques, including memory protection, virtual memory, or software fault isolation, are well suited to this purpose.

Unfortunately, such strict isolation also prevents memory sharing with foreign libraries. To enable sharing, Omniglot allows Rust to dereference pointers into foreign memory if and only if the object they reference is wholly contained inside the foreign library's valid memory regions.

Aliasing and Mutability. Rust uses strict aliasing rules to statically reason about which references can be used to safely mutate or read memory. While *unique*, mutable references can never be aliased, *shared* (aliased) references must never be modified (a rule known as *aliasing XOR mutability*).

This is problematic in the presence of foreign code: for instance, a foreign function may return references to the same memory multiple times. Since we cannot track pointer provenance through foreign memory or code, we must conservatively assume that all references derived from such pointers are mutably aliased with each other. The only way to safely support mutable aliasing in Rust is through mutual exclusion. Section 4.4 show how Omniglot introduces a zero-cost and deadlock-free mutual exclusion mechanism for references into foreign memory.

Type Safety. Saraswat defines a language as type-safe if "the only operations that can be performed on data in the language are those sanctioned by the type of the data." [36] Violating this property can break safe abstractions (Section 2.2), and introduce arbitrary undefined behavior.

Even if a foreign library is itself written using a type-safe language, its types are likely not identical in either semantics or representation to their counterparts in Rust. Indeed, translating objects between the type systems of two languages is an active area of research [32, 33, 34]. Omniglot must therefore assume that objects passed to or obtained from a foreign function do not preserve Rust's notion of type safety.

As illustrated by the example in Figure 2, Omniglot considers objects returned from a foreign library or in foreign memory *tainted* and ascribes them a special "any"-type. Omniglot can downcast such objects to a more useful type only after *validating* them. Specifically, an object is considered valid if its underlying size, alignment, bit-pattern, and other high-level invariants match those of an object of the same type created in safe Rust. For example, a well-aligned 32 bit object is a valid `char`, so long as it contains a valid, non-surrogate Unicode code point [43].

Certain Rust types carry higher-level invariants that cannot be validated solely by observing their size, alignment, and underlying bit-pattern. Examples of this are Rust references, which impose additional restrictions on their provenance and lifetimes, or typestates [2, 37]. Omniglot can never safely validate these types once written into foreign memory. Developers can instead pass symbolic representations of these objects to foreign code, referring back to a copy in memory owned by Rust.

Concurrency. Many of Rust's invariants pertain to the underlying memory state of its objects. However, when this state changes after being validated, such as through concurrent modifications by foreign code, these invariants may no longer hold, introducing unsoundness.

Thus, Omniglot requires that foreign code not modify certain values without explicit coordination with Rust. This can be accomplished through synchronization and memory isolation primitives, or by ensuring that foreign code does not execute concurrently with Rust.

4 Design

In this section we present the design of Omniglot, a framework to provide a *safe* interface to interact with foreign code. Omniglot's workflow is similar to that of Rust's conventional unsafe FFI with `rust-bindgen`, as described in Section 2.1: for a set of C header files, Omniglot generates a set of safe FFI bindings that developers can use to manipulate foreign data types and call into foreign functions.

To safely interact with an untrusted foreign library, Omniglot combines a set of inter-operating mechanisms. It features a pluggable runtime component (described in Section 4.2) responsible for loading libraries into a sandbox and isolating their memory. Omniglot further features a set of reference and validation types (Section 4.3) that capture and progressively validate the aliasing and type safety invariants described in Section 3.

However, simply applying the approach described in the previous section is not sufficient: interactions between the Omniglot runtime and its types can violate invariants that Omniglot's reference and validation types rely on. Thus, in Section 4.4 we introduce a mechanism called *scopes* that captures and enforces an additional set of *temporal invariants* to uphold the required invariants. Scopes are zero-cost, are enforced at compile-time, and statically reject code that would violate these invariants.

4.1 Threat Model

Omniglot maintains Rust's soundness across interactions with arbitrary, untrusted, and adverserial foreign code.

To achieve this goal, we assume that there is an available hardware, OS-provided, or software-based memory isolation primitive that provides strong memory protection—i.e. it can prevent a foreign library from accessing all but explicitly permitted memory regions.

Furthermore, we assume that there are OS or hardware primitives that allow restricting when foreign code can run. In particular, these primitives must be able to limit the ability of a foreign library to execute concurrently with host code, such as by preventing it from spawning and running threads that continue to run in the background, or executing in response to asynchronous signals.

We also assume that the Omniglot library, runtime implementation, Omniglot's bindings generation tool, as well as the Rust type-checker and compiler are correct. Omniglot provides a set of interfaces and types that safe Rust cannot use incorrectly. Thus, safe Rust code of the host application does not need to be trusted. However, any code that uses Rust's **`unsafe`** feature to circumvent restrictions imposed by Omniglot through the type system is considered trusted.

Omniglot can maintain Rust's soundness across interactions with adverserial libraries when using a *strong Omniglot runtime*—a runtime *enforcing* the above assumptions using strong primitives for memory isolation and restricting concurrency. We present an implementation of a strong runtime in Section 5.1. However, when these assumptions are violated, Omniglot's threat model changes. For example, a *weak runtime* may not be able to entirely prevent a foreign library from circumventing memory protection (e.g., through system calls), or restrict its access to concurrency primitives. In this case, Omniglot cannot safely run fully untrusted and adverserial foreign libraries. Instead, developers must trust that Omniglot's requirements concerning memory isolation and concurrency are upheld by the foreign library. Section 5.2 describes such a weak Omniglot runtime and further discusses its threat model implications.

Omniglot does not guarantee application-level correctness properties. For example, while it guarantees that a **`bool`** returned from a foreign function is either **`true`** or **`false`**, it does not guarantee that the foreign library computed this value correctly. Furthermore, Omniglot does not protect foreign code from operations issued by host code: for example, the host program may write to arbitary locations in foreign code's memory, including in ways that can cause foreign code to misbehave. However, assuming a strong Omniglot runtime, such misbehavior cannot compromise the host program's soundness.

Finally, Omniglot does not address timing and other side-channel attacks that might allow foreign code to infer the contents of host program memory it cannot access directly.

4.2 Sandboxing Foreign Libraries

Omniglot divides memory into regions owned exclusively by Rust, and regions delegated to foreign library instances. It uses a memory isolation primitive to uphold the host's memory safety when executing foreign libraries by restricting them to only be able to access their own, sandboxed memory regions.

As these memory isolation primitives vary between different architectures and platforms, Omniglot features a set of platform-specific runtimes. A runtime offers an interface to integrate with the core Omniglot framework. For instance, a runtime allows Omniglot and developers to to interact with a system's memory isolation primitive. It also imposes other restrictions on foreign code like limiting its access to concurrency primitives, to prevent untrusted foreign code from invalidating Rust's invariants after they have been validated by Omniglot. Omniglot's runtimes expose the following (simplified) API which host code uses to interact with foreign code:

`OGRt::new(library)` Constructs a new runtime instance and loads `library` into the sandbox.

`rt.stack_alloc(size, closure)` Allocates `size` bytes on the foreign library's stack, and invokes a closure during which this allocation is valid. It restores the previous foreign stack pointer when the closure returns.

`rt.setup_callback(callback, closure)` Prepares a callback that foreign code can trigger, and invokes a closure during which this callback is valid. This function creates an address for which, whenever foreign code jumps to it, the Omniglot runtime will execute `callback`.

`rt.invoke` Invokes a foreign function within the memory isolation sandbox. After invoke returns, the runtime guarantees that no foreign threads are running. This symbol is not called directly, and instead used by a trampolining mechanism described in Section 5.3.

The Omniglot FFI bindings for a foreign library are portable across multiple runtime implementations. These bindings resemble Rust's unsafe FFI interface, but call out to the Omniglot runtime instead of directly invoking foreign code. Nonetheless, Omniglot requires developers using a foreign library (or those writing Rust wrappers) to transform their code, for instance by using the above methods for allocating memory or callbacks that will be used by foreign code. When a foreign library attempts to access an allocation outside of its sandbox, or invokes a host callback not prepared though `setup_callback`, it triggers a fault in the memory isolation mechanism, which the Omniglot runtime catches and exposes as an `Err` value returned to Rust by `invoke`.

4.3 Communication between Rust and Foreign Code

Omniglot divides memory into mutually-exclusive regions for the host program and foreign code: it uses a memory iso-

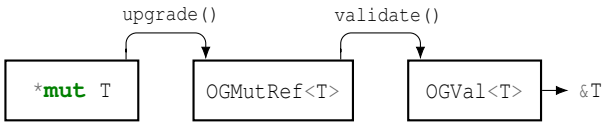


Figure 3: Omniglot captures arbitrary addresses in foreign memory as raw pointers (`*mut T`). The *upgrade* and *validate* methods transform a potentially invalid reference into writeable (`OGMutRef<T>`) and readable (`OGVal<T>`) wrappers.

lation mechanism to prevent foreign code from changing host memory. Furthermore, safe Rust is incapable of constructing arbitary references into foreign memory. While this strict division trivially upholds memory safety, it also prevents efficient transfer of data between guest (foreign) and host code.

Omniglot cannot allow foreign libraries to access memory belonging to the host program (Rust), as unrestricted, arbitrary access of foreign code to Rust's memory can break its invariants (see Section 3). Instead, Omniglot provides a mechanism that allows host code to allocate and access values in foreign memory. This is sufficient to allow host code to communicate with foreign libraries, while at the same time making it feasible to maintain Rust's soundness.

Using the runtime interface, host code can allocate objects in memory accessible to a foreign library (e.g. using `rt.stack_alloc`). Depending on the runtime, these allocations can happen on the foreign library's stack, a separate stack dedicated for such host-language allocations, or using heap memory dedicated to foreign code. These operations alone have no impact on the safety of the host program—their effects are limited to the foreign library's memory regions.

However, for Rust to safely dereference or modify memory belonging to a foreign library, we must ensure that it is contained within a *valid foreign allocation*, and that it represents a *valid instance* of its ascribed type. In this section we introduce a set of types following the typestate paradigm [2, 37] that capture validation operations and offer a *safe* API to access data shared with foreign code.

Figure 3 illustrates these *reference* and *validation* types. The first type, a raw pointer, forms the base of Omniglot's type infrastructure. It can capture arbitrary addresses in foreign memory. This pointer does not have to maintain any invariants; it may be dangling, misaligned, or alias host memory. Thus, safe Rust is not able to dereference it.

The second type, `OGMutRef<T>`, expresses pointers that refer to an active, well-aligned foreign memory allocation of sufficient size for a type `T`. It can be obtained through *upgrading* a raw pointer and checking the aforementioned invariants at runtime, by checking against the runtime-tracked set of memory regions dedicated to the foreign library. An `OGMutRef<T>` represents a useful intermediate type: when considering our model of memory safety in the presence of foreign code from Section 3, because this type must point into a *valid* allocation of foreign memory, writing to it does not violate memory safety.

However, Rust's limitations around mutable aliasing complicate this: in general, an aliased Rust reference must not be modified. As shown in Section 3, given that an `OGMutRef<T>` can be constructed from an arbitrary raw pointer supplied by foreign code, we cannot track its provenance and must instead assume that it is (partially) aliased by other references. To avoid violating Rust's aliasing constraints while still allowing `OGMutRef<T>` references to be modified, Omniglot opts out of Rust's strict aliasing model for this reference type, by representing it as the composition of two of Rust's standard library types: `&UnsafeCell<MaybeUninit<T>>`. Whereas `MaybeUninit` prevents the Rust compiler from making assumptions about the validity of values stored behind this reference, `UnsafeCell` allows values to be mutably aliased. However, an `UnsafeCell` annotation can only be used as long as every other reference to this value also carries this same annotation. As a result, Omniglot must ensure that no regular Rust references (ones not wrapped in such an `UnsafeCell`) point into foreign memory. This holds by construction, as foreign allocations are disjoint from any of Rust's allocations.

A final type, `OGVal<T>` captures pointers that refer to memory which contains an actual, dereferenceable instance of type `T`. This memory may be safely read and interpreted as its ascribed type without any risk to the soundness of the host program. Host code can attempt to *validate* an `OGMutRef<T>` into an `OGVal<T>` by inspecting its underlying bit-pattern, as described in Section 3. Omniglot performs this validation using a Rust trait, and provides implementations for many of Rust's primitive types. For any types that cannot be validated solely by inspecting their size, alignment, and bit-pattern, this trait is not implemented.

Omniglot's restrictions on concurrency also follow from the above types and reflect a fundamental constraint: so long as any validated Rust references to foreign memory exist, they must retain the invariants described in Section 3. However, if foreign code were to execute concurrently with host code, it could break these invariants at any time, such as by arbitrarily modifying any memory it can access. If the host program would access a foreign library *instance's* memory concurrently with *that instance's* execution, then the above validations could be subject to a time-of-check to time-of-use (TOCTOU) bug: after performing an *upgrade* or *validate* operation in Rust, foreign code may change its available memory allocations or write invalid values to previously validated memory locations. This would then invalidate the invariants gained in the *upgrade* and *validate* typestate transitions. Therefore, Omniglot requires that host code does not hold references into a foreign library *instance's* memory concurrently with *that instance's* execution. Omniglot's architecture can support running multiple Rust threads, as well as running multiple instances of foreign libraries concurrently.

`OGRt::new()` yields a unique allocation and access scope per library: `OGRt::new(library) -> (Rt, AllocScope, AccessScope)`

Allocation scopes prevent dangling references:
`ptr::upgrade(&AllocScope<'l>) -> OGMutRef<'l>`
`OGRt::stack_alloc(&'a mut AllocScope<'l>, Fn(&'b mut AllocScope<'m>))`

Access scopes prevent invalidating references through writes:
`OGMutRef::write(val: T, &mut AccessScope)`
`OGMutRef::validate(&'a AccessScope) -> OGVal<'a>`

`invoke` may write to memory and modify allocations in callbacks: `OGRt::invoke(&mut AllocScope, &mut AccessScope)`

Table 1: Interactions between Omniglot's runtime, reference and validation type APIs, and its *allocation-* and *access-scopes*: by binding `OGMutRef` references to their originating allocation scope, we eliminate dangling references. Validated `OGVal` references are further bound to an access scope, which expires upon foreign function execution or when foreign memory is modified.

4.4 Temporal Constraints

Omniglot's reference and validation types allow a developer to safely read and write foreign memory, as long as their underlying invariants hold (Section 3). However, even without concurrency, seemingly unrelated operations can affect these invariants: for example, writing to an `OGMutRef` may modify foreign memory previously validated into an `OGVal`. Revoking foreign allocations may cause reference or validation types to be dangling. And invoking a foreign function can both modify memory, and trigger a callback which may do any of the above.

In other words, the invariants of Omniglot's reference and validation types must be *temporally constrained*. These constraints may be quite subtle, as we can see in the following example: a foreign library yields two pointers, referencing overlapping memory that the caller interprets as an integer and a boolean respectively. The caller upgrades these pointers to `OGMutRef`s and validates one into an `OGVal<bool>`. Then, if both pointers alias each other, writing a non-boolean value to the integer reference would invalidate the invariants checked when creating the `OGVal<bool>`. As such, the boolean reference must not exist beyond the write to its aliasing integer reference. Similarly, any of these references may become invalid when Omniglot revokes foreign memory.

We could enforce these temporal constraints through mutual exclusion by using a read-write lock. Assume that we have a single lock for reading and writing a given library's memory. Then, calling `OGMutRef::validate` to produce an `OGVal<T>` could acquire a read-lock internally. This lock would be released when the resulting `OGVal<T>` goes out of scope. Additionally, both operations that can modify foreign memory, `OGMutRef::write` and `OGRt::invoke`, will acquire a corresponding write lock. Under this model, it will be impossible to retain an `OGVal<T>` across any modification of this memory—doing so will result in a deadlock.

Similarly, a read-write lock for each foreign allocation can protect upgraded references (both `OGMutRef` and `OGVal`) from memory allocation and revocation operations invalidating them. In this model, operations which may revoke memory (`ptr::stack_alloc` and `OGRt::invoke`) take a write-lock on the relevant memory allocations. Conversely, `upgrading` a pointer to create an `OGMutRef` requires a read-lock, released only when it goes out of scope. This would ensure that, if there are any outstanding references to an allocation, an attempt to revoke the allocation would result in a deadlock.

Unfortunately, *dynamic* locks do not reject incorrect code at compile time and thus place an undue burden on developers to ensure their code is deadlock-free. Such dynamic locks are also unnecessary. Given Omniglot's restrictions on concurrency—namely that Rust code does not hold references into a foreign library instance's memory concurrently with that instance's execution—we can statically enforce mutual exclusion by reasoning about data-flow patterns in programs. We design a zero-cost, compile-time enforced, static, and single-threaded locking mechanism based on lexical scopes. Omniglot uses these scopes in tandem with Rust's restriction on *aliasing XOR mutability* to implement a form of read-write mutual exclusion, and enforces it through Rust's borrow checker.

Table 1 describes Omniglot's API using scope-based exclusion. Creating a new, unique, instance of a foreign library using `OGRt::new` yields an `AllocScope` and an `AccessScope`—two branded types [19, 50] that are uniquely bound to a particular call to the constructor. *Unique* (`&mut`) borrows of these values correspond to obtaining a write-lock, and a *shared* (`&`) borrow corresponds to obtaining a read-lock. Rust prevents concurrent *unique* and *shared* borrows of the same scope. Omniglot's reference types carry a lifetime derived from the lifetime of these borrows, ensuring that they cannot outlive the `AccessScope`s and `AllocScope`s in which they have been created.

Omniglot uses these scopes to enforce its temporal constraints on `OGMutRef` and `OGVal` types. When an `OGMutRef` is validated, it captures a *shared* borrow of an `AccessScope` for an anonymous lifetime `'a`. It produces an `OGVal<'a>` that captures this lifetime. As long as this new value exists, Omniglot statically prevents calling any functions that could invalidate its invariants, such as `OGMutRef::write`, by requiring these functions to take a *unique* reference to the `AccessScope`.

We use a similar approach to maintain the constraints for `OGMutRef`. Upgrading a pointer captures a *shared* reference to an `AllocScope`, while any operations that may revoke memory, and thus endanger the invariants of this type, take a *unique* `AllocScope` reference.

In practice, Omniglot can be slightly more permissive about letting `OGMutRef`s outlive certain revocations of memory. By combining an allocation with its corresponding revocation

into a single function—`stack_alloc`—Omniglot allows references that were created before this allocation to also outlive its revocation. In other words, while Omniglot must prevent any reference created in a call to `stack_alloc` from escaping its inner closure, those created in outer scopes can remain live. Omniglot does so by separating the lifetime that an `AllocScope` is active for (`'a`), from the lifetime of its underlying memory allocation (`'l`), as shown in Table 1.

Finally, invoking a foreign function can both modify memory, and trigger a callback that can revoke allocations, and thus captures *unique* references to both allocation and access scopes.

4.5 Omniglot API Walkthrough

Listing 3 illustrates how all of Omniglot's mechanisms interact to provide a safe interface to an arbitrary, untrusted foreign library. On line 3, this example loads a fictional compression library (`libcompress.so`) into an Omniglot runtime. Constructing a runtime and loading a library yields four return values: a runtime object `rt` exposing the API shown in Section 4.2, a library shim `lib` featuring wrapper methods for all foreign library functions, and allocation and access scope marker values as described in Section 4.4.

In this example, we use the foreign library to compress an array of 4 bytes. On line 7 we use the runtime's `stack_alloc` function to place a new allocation of type `[u8; 4]` on the foreign library's stack. Because allocating memory can change the set of allocations accessible to foreign code, it interacts with Omniglot's allocation scopes: `stack_alloc_t` (to allocate space for a generic type `T`) obtains a *unique* reference to the outer allocation scope marker `oalloc` on line 8 and holds onto it through line 31. This *unique* borrow prevents new *upgrade* operations (potentially on the new, stacked and ephemeral allocation) from using this outer allocation scope. In turn, `stack_alloc_t` provides a new *inner* allocation scope `ialloc` which is valid for the duration of this new allocation. Any *upgrade* operations will thus use this new allocation scope instead; we say the *active* allocation scope changes (switching from a solid to a dashed line in Listing 3).

We write uncompressed data to this newly allocated buffer on line 10. As illustrated in Section 4.4, writing any foreign memory may invalidate other references. Thus, this operation obtains a *unique* reference to the access scope marker: it closes the previous scope, and immediately re-opens a fresh access scope (dotted line in Listing 3). Finally, we invoke the foreign library's `compress` function on line 15, which executes it in the Omniglot runtime's sandbox. Invoking a foreign function may both change foreign memory *and* change the set of foreign allocations (in a callback), and thus requires a *unique* reference to both allocation and access scopes.

Listing 3 demonstrates scope-based enforcement of Omniglot's temporal constraints: for instance, line 12 validates the allocated buffer into an `OGVal<[u8; 4]>` reference.

```
1       // Load a library into a runtime, creating
2       // allocation and access scope markers:
3       let (rt, lib, mut oalloc, mut access) =
4         Rt::new("libcompress.so");
5
6       // Allocate a [u8; 4] on the foreign stack
7       rt.stack_alloc_t::<[u8; 4]>(
8         &mut oalloc, |array, ialloc| {
9           // array: OGMutRef<'_, [u8; 4]>
10          array.write([0, 1, 2, 3], &mut access);
11
12          let validated = array.validate(&access);
13          println!("{:?}", validated);
14
15          let compressed_len = lib.compress(
16            buf: array.as_ptr(),
17            length: 4,
18            &mut ialloc, &mut access).validate();
19
20          // Would not compile, as `validated` is
21          // bound to the previous access scope:
22          println!("{:?}", validated);
23
24          let upgraded = array.as_ptr()
25            .upgrade_slice(compressed_len, &ialloc);
26          let revalidated = upgraded.validate(&access);
27          println!("{:?}", revalidated);
28
29          // `array` and `upgraded` cannot escape
30          // closure, capture the `ialloc` scope.
31        }
32      )
33    Active Access Scope
34  Active Allocation Scope
```

Listing 3: A basic Omniglot wrapper around a compression library, with its allocation and access scopes visualized. We create an Omniglot runtime, allocate memory, invoke a foreign function, and validate its returned result. Dashed and dotted lines represent the active and open allocation and access scopes, respectively. Each mutable borrow of a scope marker closes the current and re-opens a new scope, causing all references bound to the previous scope to be inaccessible.

However, this reference is bound to the access scope closed on line 15. Thus, any attempts to access it (as on line 22) will result in a compile-time error. Instead, correct bindings create a new `OGMutRef<[u8]>` reference over the compressed data length (line 25), and then re-validate it against the currently open access scope (line 26).

5 Implementation

We implement Omniglot and its two runtimes in a set of Rust crates of approx. 7000 LoC. We further extend `rust-bindgen` by 860 LoC to automatically generate Omniglot-compatible bindings from C header files. Our implementation is available on GitHub: https://github.com/

omniglot-rs/omniglot.

In this Section, we first present our two prototype Omniglot runtime implementations for the Tock operating system (using RISC-V PMP) and Linux userspace (using x86 Memory Protection Keys) respectively. As these runtimes use different isolation primitives and run in different environments, we classify them into *strong* and *weak* runtimes as per our threat model (Section 4.1), and discuss which—if any—assumptions the Omniglot runtimes make on the behavior of foreign libraries.

Finally, in Section 5.3 we present Omniglot's `invoke` trampoline. While Omniglot is implemented in the type-system of an unmodified Rust compiler, Rust normally does not allow libraries to interpose on calls to foreign functions. Omniglot's `invoke` trampoline makes Omniglot practical by allowing it to switch to a foreign protection domain on calls to foreign functions. In contrast to existing approaches like `libffi`[6], we do so with minimal overhead, by using the Rust compiler's knowledge of the foreign function's ABI to statically generate code for loading function arguments into registers and on the stack.

5.1 Omniglot for the Tock Operating System

OG_{PMP} is an implementation of Omniglot for the Tock embedded operating system kernel [21], utilizing the RISC-V Physical Memory Protection (PMP) unit as its memory isolation primitive. The Tock kernel is itself written in Rust and used for security critical applications, such as in the firmware of the OpenTitan silicon root-of-trust or Microsoft's Pluton security controller embedded in modern x86 CPUs [49]. Because Tock heavily relies on Rust's safety for enforcing isolation between mutually distrustful applications and must, for instance, integrate existing and certified cryptography libraries, it is an ideal target for Omniglot.

The RISC-V PMP is a secure memory isolation primitive and thus enables OG_{PMP} to provide strong soundness guarantees. In particular, OG_{PMP} loads foreign libraries into protection domains similar to Tock processes, running at a lower privilege level, and without any memory of the Tock kernel accessible to them. It executes library functions by loading their arguments into registers and onto a dedicated stack in foreign memory, and context-switching to a lower-privilege mode. Libraries can transition back into the Tock kernel (which corresponds to Omniglot's host program) by one of three mechanisms: returning from the function invoked by the kernel, invoking a registered callback, or producing a fault—in which case Omniglot returns an error to host code. Notably, there is no standard system call interface accessible to foreign code, and OG_{PMP} does not expose any concurrency primitives.

Because OG_{PMP} fully isolates memory belonging to a foreign library, does not provide any means to circumvent this isolation, and does not expose any concurrency primitives to the foreign library, it is considered a *strong* Omniglot runtime. It can run arbitrary, untrusted and adverserial foreign libraries, while maintaining the soundness of the Tock kernel.

[6] https://sourceware.org/libffi/

5.2 Omniglot for Linux Userspace

On the other end of the spectrum, OG_{MPK} brings Omniglot to Linux userspace applications on desktop- and server-class systems. OG_{MPK} utilizes x86 Memory Protection Keys (MPK), a recent isolation mechanism built around the idea of assigning virtual memory pages one of 16 *protection keys*. A core-local and userspace-writeable register controls which protection keys are accessible at any given time. Notably, MPK does not require a system call to switch protection domains, making domain transitions fast.

OG_{MPK} uses *link-map lists* to load conventional shared libraries into an isolated namespace. It then uses MPK to assign all pages associated with a foreign library—and its transitive dependencies—a protection key. When running functions of the foreign library, OG_{MPK} disables access to all memory pages not assigned this protection key—making the memory of host code and other foreign libraries inaccessible.

While MPK is a popular tool for intra-process memory isolation [4, 16, 20, 23, 38, 46], in particular due to its low overheads, it presents some challenges for security: for instance, Connor et al. show that MPK-based isolation can be circumvented through system calls, memory mappings, signal delivery, and race conditions [8]. As Omniglot's focus is on maintaining Rust's soundness assuming a strong memory isolation primitive, we do not address the inherent security properties of MPK in this work. Therefore, OG_{MPK} is a *weak* Omniglot runtime.

Our implementation of OG_{MPK} does override some symbols such as `alloc` and `free` to use a custom, memory-isolation aware allocator. However, actively malicious libraries can circumvent OG_{MPK}'s memory isolation by using system calls like `mmap`. Developers will need to validate that foreign libraries do not perform such operations, or employ `seccomp-bpf`-based mitigations as presented in ERIM [46]. Similarly, OG_{MPK} does not prevent foreign libraries from accessing concurrency primitives. Developers must ensure that libraries do not register signal handlers and run background threads concurrently with host code execution, such as by manually inspecting the foreign libraries, disallowing them from using certain system calls, or inspecting the state of the program at runtime.

Despite these weaker guarantees, OG_{MPK} is still useful to isolate libraries that are not assumed to be actively malicious. Its reference and validation types can prevent many cases of improper mutable aliasing and can ensure that values used by Rust conform to its strict validity requirements.

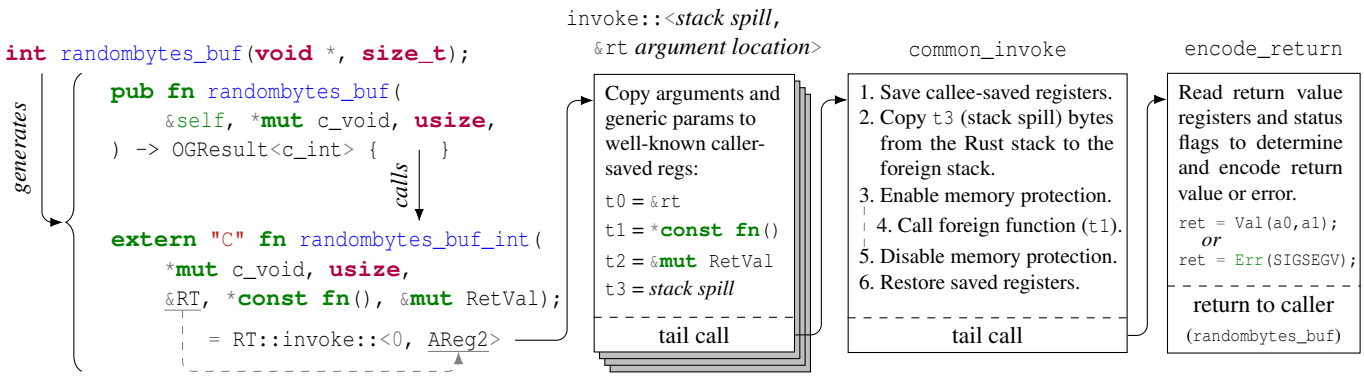


Figure 4: Omniglot wraps foreign functions by creating an internal, unsafe FFI binding and aliasing it to the Omniglot runtime's generic invoke symbol. The runtime preserves the Rust compiler's argument registers and stack.

5.3 The `invoke` Trampoline

An Omniglot runtime is responsible for ensuring that any foreign code runs within its sandbox and, in turn, all callbacks execute in Rust's context, with access to both Rust and foreign memory. Implemented naïvely, however, this could prevent a foreign function from accessing its parameters or writing its return value, as they might be spilled on the host's stack. Omniglot solves this issue by copying this memory to and from the foreign stack. This poses a challenge: how can the Omniglot runtime know which memory to copy? Importantly, this depends not only on the platform's calling convention, but also on each individual function's signature.

To solve this problem, we introduce a generic `invoke` trampoline mechanism, illustrated in Figure 4. Rather than manually placing function arguments into their appropriate registers and stack offsets, Omniglot relies on the Rust compiler's knowledge of the C calling convention (as opposed to using a dynamic approach like `libffi`). In particular, our modified `rust-bindgen` generates a new **`extern "C"`** Rust function for each C function binding, which expects the same arguments as the original C function. When calling this generated symbol, the compiler will therefore correctly lay out arguments according to the C calling convention. This process is captured on the left column of Figure 4, for an example C function `randombytes_buf`.

We must now switch protection domains before executing the function. This switch is implemented by a single, generic `invoke` function, implemented once per runtime. The invoke function executes some argument-specific setup code, some common protection setup code, and then dispatches to the target C FFI function. We must ensure that `invoke` is called whenever a C FFI function is invoked. To do this, we alias the C-ABI function symbol for the FFI call directly to the generic `invoke` symbol. This has the effect of translating calls to our generated FFI binding (**`fn`** `randombytes_buf_int` in Figure 4) to calls to `invoke`, allowing the protection domain setup to proceed. This technique effectively allows us to insert custom protection instructions directly into a function prologue, *without modifying* the foreign library or Rust compiler, and without requiring Omniglot users to write any boilerplate code. It does however introduce its own challenge: within the body of the `invoke` function, we do not automatically know which function should be invoked!

To solve this, we pass a combination of runtime and generic arguments to the invoke function (which causes the compiler to monomorphize it into concrete instantiations bound to its generic parameters). Taken together, these arguments allow the monomorphized `invoke` routine to find the function to be invoked, locate the arguments for that function, find the current foreign stack pointer, and finally store the function's returned value. These arguments can be seen in the example in Figure 4. The first two arguments to `randombytes_buf_int` are the arguments for the target C function. Of the remaining invoke-specific arguments, the `&rt` parameter is used to lookup library-specific information (like the location of the foreign stack pointer), the **`*const fn`**`()` function pointer stores the location of the original foreign function (**`int`** `randombytes_buf()`), and the `RetVal` parameter is a container for the eventual return value. The generic argument `AReg2` tells invoke where to find these extra arguments[7].

6 Evaluation

This section evaluates Omniglot using multiple libraries across different application domains and two memory isolation primitives. With our evaluation we show the following:

Omniglot is general. It is able to safely encapsulate libraries with different interface paradigms, across a wide range of application domains. Omniglot further supports a wide range of systems, from high performance desktop and

[7] Its accompanying integer is used to indicate arguments' offset into the stack, whenever arguments need to stack-spill.

Library	Runtime	#invoke	#callback	#upgrade	#validate	unsafe	isolation only		Omniglot	
`CryptoLib`	PMP	14	0	10	2	9105 µs	9145 µs	(+0.44%)	9145 µs	(+0%)
`LittleFS`	PMP	7	0	0	7	3115.3 µs	3742.8 µs	(+20.1%)	3764 µs	(+0.5%)
`LwIP`	PMP	3	1	2	5	51.74 µs	78.71 µs	(+52.1%)	81.4 µs	(+3.4%)
`Brotli`	MPK	2	0	0	4	3.12 ms	3.14 ms	(+0.6%)	3.14 ms	(+0%)
`libsodium`	MPK	1	0	0	2	51.61 µs	53.41 µs	(+3.4%)	53.41 µs	(+0%)
`libpng`	MPK	5	15	16	3	352.98 µs	397.93 µs	(+12.7%)	401.25 µs	(+0.8%)

Table 2: Omniglot overheads for a diverse set of libraries, across two runtimes. We show the number of Omniglot operations per benchmark iteration, and compare its overheads to an unsafe baseline, and memory isolation without Omniglot's runtime checks.

server systems to highly resource-constrained microcontrollers, as well as userspace and kernel environments.

Omniglot is fast. Omniglot's ability to perform zero-copy accesses into foreign memory allows it to efficiently convey data between host and foreign code. As we will show in Section 6.2, this allows Omniglot to outperform related work that delivers similar safety guarantees.

We lift the enforcement of Omniglot's mutual exclusion requirement into Rust's type-system, turning it into a zero-cost abstraction. In this evaluation, we further show that the Rust compiler is able to optimize out validation in many cases and, where it cannot, those overheads are small.

Because Omniglot's overheads are primarily governed by its underlying memory isolation mechanism, it delivers competitive performance to existing systems that do not maintain other safety-critical invariants such as aliasing or type safety.

We consider two case studies: Section 6.1 shows how Omniglot can be used to efficiently integrate existing, unsafe foreign libraries into an operating system kernel. Section 6.2 evaluates Omniglot's support for loading and interacting with widely-used userspace libraries. Table 2 summarizes the results of our measurements across both case studies. Finally, Section 6.3 provides a breakdown of Omniglot's overheads through a series of microbenchmarks.

Experiment Setup. We evaluate two different Omniglot runtimes on different systems. User-space Linux evaluations run on a CloudLab Wisconsin `c220g5` node with two Intel Xeon Silver 4114 CPUs at 2.2 GHz, and frequency scaling and hyperthreading disabled, using rustc 1.84.0-nightly and Linux 5.15.0. The benchmark process is pinned to a single CPU core. Kernel evaluations run using a port of the Tock operating system[8], on a ChipWhisperer CW310 FPGA with the OpenTitan EarlGrey silicon root-of-trust design[9] and a RISC-V `rv32imc` CPU at 24 MHz.

[8] Based on https://github.com/tock/tock, git revision `1967fa7`.
[9] https://github.com/lowrisc/opentitan, git revision `bbbd9b0`.

6.1 Case Study: Tock Kernel

Tock is an operating system commonly used in security-critical resource-constrained systems. Its kernel is written in Rust and its design goes to significant lengths to minimize the use of unsafe Rust. Nonetheless, practical deployments of Tock often rely on foreign libraries to implement critical operations. We evaluate OG_{PMP} on three such libraries, across a range of application domains and interface paradigms.

We evaluate the OpenTitan CryptoLib library by encapsulating its HMAC-SHA256 algorithm. The results in Table 2 show the execution time, as well as the number of Omniglot operations performed in a single HMAC operation over 4 kB of data, carried out in batches of 512 B. Because this benchmark requires Omniglot to copy data into the CryptoLib's memory, enabling PMP-based memory isolation increases execution time by 0.44%. Omniglot's runtime checks (during *upgrade* and *validate*) introduce no measureable overhead in this case. In this benchmark, Omniglot tracks only a single allocation in the foreign library, its *HMAC context*, making upgrade operations fast. Furthermore, Omniglot requires developers to *validate* the returned *HMAC* tag. However, because this tag is a simple byte-array, and a `u8` integer is an unconditionally valid type, the Rust compiler is able to optimize such validation away entirely.

Additionally, Omniglot supports libraries that maintain state between calls. We evaluate OG_{PMP} on LittleFS, a filesystem library. We repeatedly format the filesystem, mount it, create and open a file, write 1 kB UTF-8 code points to it, read the data back, and then close the file. In this benchmark, we compare the contents of the file with the original string. To do this, Omniglot forces us to validate that the returned buffer is a valid Rust string (*i.e.*, valid UTF-8). This operation is linear in the amount of data to validate, and explains OG_{PMP}'s overhead of 0.5% compared to memory isolation.

Finally, we demonstrate Omniglot's ability to handle callbacks. We do so by evaluating OG_{PMP} with the LwIP embedded network stack. Our benchmark sends an ICMP echo request by allocating a new buffer, copying the packet, and injecting it into the network stack. In response, LwIP invokes a

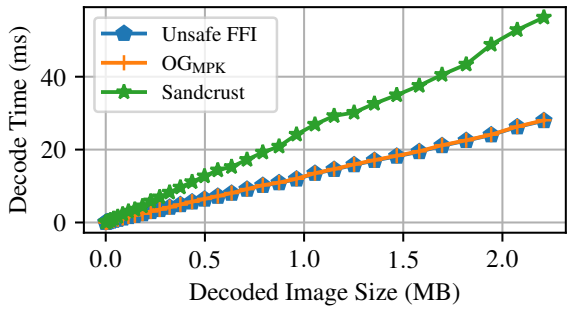


Figure 5: Execution time of decoding a PNG image, as function of the decoded image size. We compare OG_{MPK} against an unsafe FFI baseline and Sandcrust. Omniglot's ability to perform zero-copy accesses to foreign memory allows it to outperform Sandcrust, which requires serialization and copying.

(a)	Setup		Invoke	
	PMP	MPK	PMP	MPK
Unsafe	0.17 µs	1.49 ms	0.17 µs	13.74 ns
Omniglot	73.88 µs	4.08 ms	6.57 µs	98.90 ns
Sandcrust		1.79 ms		10.87 µs
Tock Upcall			56.20 µs	

(b)	Upgrade		Callback	
allocs / CBs	PMP	MPK	PMP	MPK
1	0.6 µs	32.30 ns	8.66 µs	230.1 ns
8	1.7 µs	164.1 ns	9.82 µs	364.3 ns
64	9.4 µs	1.10 µs	18.22 µs	1.32 µs

(c) Validate	64 B `u8`	8 kB `u8`	64 B `str`	8 kB `str`
PMP	0.22 µs	0.23 µs	2.74 µs	161.5 µs
MPK	1.67 ns	1.70 ns	359.6 ns	70.94 µs

Table 3: Omniglot's overheads broken down by operation.

callback previously registered with OG_{PMP}, passing a pointer to the ICMP echo response packet. OG_{PMP} needs to upgrade this pointer to a reference to read it, adding an overhead of 3.4% over memory isolation. Using this reference, it is able to directly access and read this response in foreign memory without copies. Nonetheless, the small packet size of ICMP echo requests result in a pronounced memory isolation switching overhead for invocations and callbacks of 52.1%.

6.2 Case Study: Linux Userspace

Our OG_{MPK} implementation makes Omniglot usable for isolating conventional userspace libraries. To demonstrate this, we evaluate OG_{MPK} on three libraries performing data compression, cryptographic algorithms, and image encoding. We also compare the performance and overheads of Omniglot to Sandcrust [18], an IPC and serialization-based approach to isolate untrusted code within a Rust program.

Brotli, a data compression library, and Sodium, a cryptography library, feature similar interface paradigms. Specifically, they perform "one-shot" computations on some input buffer, producing a corresponding output buffer. For Brotli, we benchmark compressing and decompressing 1 kB of UTF-8 encoded English text. Following decompression, we check that the decompressed text equals our input. Similar to LittleFS, Omniglot forces us to perform Rust's string validation for this operation. The overhead of this validation is masked by the long compression routine. For Sodium, we hash 32 kB of random inputs. As the output is a simple byte-array, validation is optimized out by the compiler.

Lastly, we use an encapsulated `libpng` to decode reference images with varying dimensions. Figure 5 shows the runtime of these operations compared to both an unsafe FFI and Sandcrust for a range of output sizes, whereas Table 2 shows the runtime for a 23 kB image (Omniglot's relative overheads decrease in the image size). `libpng` uses a callback to progressively read image data from the host program's memory. It also expects callers into the library to use `setjmp` for handling errors internal to the library. Omniglot does not support this directly, as it expects every function to return normally. We thus create a wrapper in C around these functions, converting a `longjmp` into an error return value.

Figure 5 shows that OG_{MPK}'s performance is close to that of the native, unsafe FFI bindings, while Sandcrust incurs additional overheads increasing with the decoded image size. This is because Sandcrust uses IPC and serialization to shuttle data between host and foreign memory. While this gives Sandcrust strong soundness properties, similar to those of Omniglot, it imposes significant overheads.

6.3 Microbenchmarks

We quantify Omniglot's runtime overheads through a comprehensive set of microbenchmarks. Subtable 3 (a) illustrates the time required to instantiate an Omniglot runtime (*Setup*, measured as time it takes to start the process and run a single foreign function until it returns), and to invoke a single foreign function which returns immediately (*Invoke*). The table shows OG_{PMP}'s *hot invoke* path, where the RISC-V PMP is already configured. When this is not the case, for instance by switching to a different library or process, OG_{PMP} instead incurs an overhead of 10.49 µs (adding 3.92 µs for PMP re-configuration to the 6.57 µs for the *hot invoke* path). We compare Omniglot's overheads across its two runtimes with Rust's unsafe FFI as a baseline. We further compare OG_{MPK} to Sandcrust's IPC-based approach [18], and OG_{PMP} to conventional Tock process isolation.

Omniglot must keep track of the set of allocations and callbacks registered at runtime. Subtable 3 (b) shows that Omniglot's overheads for *upgrade* operations and callbacks are linear in the number of such elements tracked. OG_{PMP} has

higher base-overhead for callbacks compared to OG_{MPK}, as they are implemented similar to a context switch.

Finally, Subtable 3 (c) shows that Omniglot's *validate* is linear in the amount of data validated. However, when types are unconitionally valid, such as fixed-width integer types, the Rust compiler is able to optimize out validation entirely. We can see this in the difference between validating 64 B and 8 kB for `u8` integers.

7 Related Work

Omniglot builds upon a significant body of work in the programming languages, systems, and security communities. The security implications of foreign function interfaces have only recently been recognized as their own class of vulnerabilities. For instance, Mergendahl et al. establishes the concept of *Cross-Language Attacks*, an entirely new set of attack vectors [26]. One way to reduce these vulnerabilities is through static analysis techniques, such as proposed by Li et al. [22].

Omniglot employs intra-process isolation mechanisms similar to a range of contributions [3, 4, 14, 16, 17, 18, 23, 29, 35, 38, 46, 52]. However, each of these require expensive operations such as copying and serialization to maintain soundness, or do not consider it at all. Omniglot is the first system that maintains soundness across interactions with foreign code while supporting zero-copy accesses into foreign memory.

We design a type infrastructure to model and mediate safe interactions with foreign code. This closely relates to work on *Linking Types*, which seek to extend the type system of a host language to encode the behaviors and semantics native to guest (foreign) code [13, 32, 33, 34, 45]. Existing work on linking types is focused primarly on the core semantic questions or type system development, attempting to express foreign semantic guarantees within a host's type system. However, these contributions assume that foreign code is internally correct. Omniglot can safely interact with libraries without reasoning about whether they are correct.

Finally, verification based approaches like eBPF [24, 28], promise safe interactions with untrusted code, with only few or no runtime checks. While these tools are growing increasingly capable and are used in complex settings such as storage functions [51] and network functions [27], they cannot encapsulate arbitrary, untrusted and unmodified libraries. Recent work proposes augmenting the static nature of such verification-based approaches with runtime checks to regain more expressiveness [12].

8 Discussion & Future Work

Deeper integration with foreign languages. Omniglot can isolate untrusted foreign libraries without making any assumptions about their behavior. While this is generally desirable, by exploring deeper integrations with foreign languages we might be able to reduce Omniglot's overheads even further. Additionally, while Omniglot as presented does not prevent a host program from violating soundness in foreign libraries, relying on specific properties of the language in which foreign code was written may allow Omniglot to enforce safety and soundness in foreign code as well.

Other host languages. In this paper, we focus on implementing Omniglot specifically for Rust, a popular choice for safe systems programming today. However, other safe languages have a similar need for sound interactions with foreign code. Future work should explore how Omniglot's concepts can be mapped to other safe languages.

Better host language bindings. Finally, Omniglot is not API-compatible with Rust's existing, unsafe C FFI. While our experience shows that most libraries can be integrated with the Omniglot bindings in a highly mechanical way, Omniglot does not support automatically transforming existing, unsafe usages of foreign libraries to Omniglot's safe APIs. We do argue that Omniglot's API does not place undue burdens on developers though—in fact, even for correct bindings with Rust's unchecked FFI, developers would still need to perform the validations that Omniglot enforces.

9 Conclusion

Software written in type-safe and memory-safe programming languages has to interact with components written in foreign, unsafe languages. A single violation of the language's safety invariants can infringe upon the application's overall safety.

This paper presents Omniglot, a system that preserves soundness of host code in the presence of untrusted foreign code. We demonstrate Omniglot's generality and low overheads by evaluating it on two systems and a diverse set of libraries. We show that Omniglot can safely encapsulate foreign libraries with minimal overheads.

Acknowledgments

We thank our shepherd, Andrea Lattuada, for his in depth feedback as well as our anonymous reviewers for their invaluable feedback. We also thank Nicolas Koh, Natalie Popescu, Shai Caspin, and Gongqi Huang for suggestions on early versions of the work as well as drafts of the paper. This work was supported by the National Science Foundation under Grants No. CNS-2443589, CNS-2321723, and TIP-2303639. Leon Schuermann was supported through a Francis Robbins Upton Fellowship in Engineering. The opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the authors institutions.

References

[1] CVE-2023-30624 in the Rust Wasmtime crate: Wasmtime has undefined behavior in Rust runtime functions. URL https://nvd.nist.gov/vuln/detail/CVE-2023-30624. Accessed 11/01/2024.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587684. doi:10.1145/1639950.1640073.

[3] Hussain M. J. Almohri and David Evans. Fidelius Charm: Isolating unsafe Rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 248–255, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356329. doi:10.1145/3176258.3176330.

[4] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. TRust: A compilation framework for in-process isolation to protect safe Rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, Anaheim, CA, 08 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/bang.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995. ISSN 0163-5980. doi:10.1145/224057.224077.

[6] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, 11 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/boos.

[7] Thomas Claburn. Microsoft is busy rewriting core Windows code in memory-safe Rust, 04 2023. URL https://www.theregister.com/2023/04/27/microsoft$_{w}$indows$_{r}$ust/. Accessed 10/07/2023.

[8] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, 8 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/connor.

[9] Jonathan Corbet. A first look at Rust in the 6.1 kernel. URL https://lwn.net/Articles/910762/. Accessed 04/19/2024.

[10] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, page 69–80, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136471. doi:10.1145/780732.780743.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, 07 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/duplyakin.

[12] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 249–264, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi:10.1145/3694715.3695950.

[13] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 62–72, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi:10.1145/1065010.1065019.

[14] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. Friend or foe inside? Exploring in-process isolation to maintain memory safety for unsafe Rust. In *2023 IEEE Secure Development Conference (SecDev)*, pages 54–66, 2023. doi:10.1109/SecDev56634.2023.00020.

[15] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. ISSN 0163-5980. doi:10.1145/1243418.1243424.

[16] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-Safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 132–148, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi:10.1145/3492321.3519582.

[17] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi:10.1145/3498688.

[18] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 51–57, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351539. doi:10.1145/3144555.3144562.

[19] John Launchbury and Simon L Peyton Jones. State in Haskell. *LISP and Symbolic Computation*, 8:293–341, 1995. doi:10.1007/BF01018827.

[20] Hanwen Lei, Ziqi Zhang, Shaokun Zhang, Peng Jiang, Zhineng Zhong, Ningyu He, Ding Li, Yao Guo, and Xiangqun Chen. Put your memory in order: Efficient domain-based memory isolation for WASM applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 904–918, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi:10.1145/3576915.3623205.

[21] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi:10.1145/3132747.3132786.

[22] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. Detecting cross-language memory management issues in Rust. In *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*, pages 680–700, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-17142-0. doi:10.1007/978-3-031-17143-7_33.

[23] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe Rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 234–245, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi:10.1145/3377811.3380325.

[24] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[25] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. A study of undefined behavior across foreign function boundaries in Rust libraries, 2024. URL https://arxiv.org/abs/2404.11671.

[26] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL https://www.ndss-symposium.org/ndss-paper/auto-draft-259/.

[27] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021. doi:10.1109/TNSM.2021.3055676.

[28] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packer filter: an efficient mechanism for user-level network code. *SIGOPS Oper. Syst. Rev.*, 21(5):39–51, November 1987. ISSN 0163-5980. doi:10.1145/37499.37505.

[29] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[30] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX

Association, 11 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram.

[31] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, Savannah, GA, 11 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda.

[32] Daniel Patterson and Amal Ahmed. Linking types for multi-language software: Have your cake and eat it too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 12:1–12:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.SNAPL.2017.12.

[33] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. Semantic soundness for language interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 609–624, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi:10.1145/3519939.3523703.

[34] Daniel Patterson, Andrew Wagner, and Amal Ahmed. Semantic encapsulation using linking types. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2023, pages 14–28, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702990. doi:10.1145/3609027.3609405.

[35] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe Rust safe with Galeed. In *Annual Computer Security Applications Conference*, ACSAC '21, pages 824–836, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385794. doi:10.1145/3485832.3485903.

[36] Vijay Saraswat. Java is not type-safe, 08 1997. URL https://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html. Accessed 11/29/2024.

[37] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929.

[38] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with Intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, pages 143–156, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375542. doi:10.1145/3381052.3381326.

[39] Synopsys, Inc. The heartbleed bug, 2020. URL https://heartbleed.com/. Accessed 10/07/2023.

[40] The Rust Contributors. The Rust reference – textual types, 2023. URL https://doc.rust-lang.org/reference/types/textual.html. Accessed 10/07/2023.

[41] The Rust Contributors. The Rust reference – behavior considered undefined, 2023. URL https://doc.rust-lang.org/reference/behavior-considered-undefined.html. Accessed 10/07/2023.

[42] The Rust Contributors. The Rustonomicon, 2023. URL https://doc.rust-lang.org/nomicon/. Accessed 10/07/2023.

[43] The Rust Contributors. The Rust core library – primitive type char, 2024. URL https://doc.rust-lang.org/std/primitive.char.html. Accessed 04/10/2024.

[44] The Rust Contributors. Unsafe code guidelines reference – glossary, 2024. URL https://rust-lang.github.io/unsafe-code-guidelines/glossary.html. Accessed 12/02/2024.

[45] Alexi Turcotte, Ellen Arteca, and Gregor Richards. Reasoning about foreign function interfaces without modelling the foreign language. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:32, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-111-5. doi:10.4230/LIPIcs.ECOOP.2019.16. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.16.

[46] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, 8 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[47] Jeffrey Vander Stoep. Memory safe languages in Android 13, 12 2022. URL https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html. Accessed 10/07/2023.

[48] White House Office of the National Cyber Director (ONCD). Press release: Future software should be memory safe, 02 2024. URL https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/. Accessed 06/05/2025.

[49] Christof Windeck. Microsoft security controller Pluton is also coming to Intel Core. *Heise*. URL https://www.heise.de/en/news/Microsoft-security-controller-Pluton-is-also-coming-to-Intel-Core-9833954.html. Accessed 12/10/2024.

[50] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: separating permissions from data in Rust. *Proc. ACM Program. Lang.*, 5(ICFP), 08 2021. doi:10.1145/3473597.

[51] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, 07 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zhong.

[52] Tianyang Zhou. Fine-grained library sandboxing for Rust ecosystem, 2023. URL https://escholarship.org/uc/item/5kq7s1jj.

A Artifact Appendix

Abstract

Omniglot is a framework to safely interact with libraries written in foreign languages from within Rust. This artifact contains our research prototype implementation of the Omniglot framework for Linux userspace (x86 MPK) and the Tock embedded operating system kernel (RISC-V PMP). We also include scripts to install any required dependencies for building and running Omniglot, and scripts to generate the evaluation results as presented in the paper. More details are available in the artifact's `README.md` file.

Scope

This artifact demonstrates Omniglot's ability to work across multiple platforms, such as in Linux userspace or resource-constrained embedded systems, and its support for a range of libraries. It allows readers to evaluate the performance and overheads of Omniglot, and validate the claims made in the paper. Specifically, it can reproduce the results of the following experiments:

- Omniglot's overheads for a set of libraries (for both OG_{MPK} and OG_{PMP}, Table 2).
- Omniglot's ability to perform zero-copy accesses into foreign memory (Figure 5).
- Omniglot's overheads broken down by operation (for both OG_{MPK} and OG_{PMP}, Table 3).

Contents

The artifact contains the implementation of Omniglot, its two runtime implementations, and other dependencies. The Omniglot framework is implemented as a set of Rust crates, split across multiple repositories available on GitHub in the `omniglot-rs` organization (`https://github.com/omniglot-rs`):

`omniglot`: Contains shared infrastructure and implementations of the reference and validation types (see Section 4).

`omniglot-mpk`: Contains the implementation of the OG_{MPK} runtime for Linux userspace, as well as bindings and benchmarks for the libraries evaluated using this runtime (see Section 6).

`omniglot-tock`: Contains the implementation of the Omniglot runtime for the Tock operating system kernel on RISC-V platforms (OG_{PMP}). It also contains bindings and benchmarks for the libraries which we evaluate using this runtime (see Section 6).

`rust-bindgen`: This repository holds our fork of the `rust-bindgen` utility for generating Omniglot FFI bindings from a C header file.

Hosting

The Omniglot framework is maintained under the `omniglot-rs` GitHub organization: `https://github.com/omniglot-rs`.

We provide instructions for reproducing the artifact evaluation through a Zenodo record under the following identifier: doi:10.5281/zenodo.15602886. This record also contains scripts and archived versions of the Omniglot components and dependencies required for reproducing the evaluation results of the paper.

Requirements

Each Omniglot runtime is designed to be evaluated on a specific platform:

OG_{MPK}: The OG_{MPK} evaluations must be run on a system with x86 MPK support. We design our evaluation scripts to run on a CloudLab [11] `cc220g5` node. The included instructions demonstrate how to set up a CloudLab experiment for evaluating OG_{MPK}.

We caution against executing the evaluation scripts on a developer's personal machine, as it installs system-wide dependencies and attempts to make persistent changes to the machine's bootloader configuration.

OG_{PMP}: The OG_{PMP} evaluations expect to be run on a ChipWhisperer CW310 FPGA running the OpenTitan EarlGrey SoC design, built from the OpenTitan repository at revision `bbbd9b0`.

Installation and Experiment Workflow

The artifact's `README.md` file contain detailed instructions on how to prepare the OG_{MPK} and OG_{PMP} evaluation platforms and reproducing the evaluation results.



KRR: Efficient and Scalable Kernel Record Replay

Tianren Zhang
SmartX

Sishuai Gong
Purdue University

Pedro Fonseca
Purdue University

Abstract

Modern kernels are large, complex, and plagued with bugs. Unfortunately, their large size and complexity make kernel failures very challenging for developers to diagnose since failures encountered in deployment are often notoriously difficult to reproduce. Although record-replay techniques provide the powerful ability to accurately record a failed execution and deterministically replay it, enabling advanced manual and automated analysis techniques, they are inefficient and do not scale with modern I/O-intensive, concurrent workloads.

This paper introduces KRR, a kernel record-replay framework that provides a highly efficient execution recording mechanism by narrowing the scope of the record and replay boundary to the kernel. Unlike previous record-replay whole-stack approaches, KRR adopts a split-recorder design that employs the guest and the host to jointly record the kernel execution. Our evaluation demonstrates that KRR scales efficiently up to 8 cores, across a range of different workloads, including kernel compilation, RocksDB, and Nginx. When recording 8-core VMs that run RocksDB and kernel compilation, KRR incurs only a $1.52\times \sim 2.79\times$ slowdown compared to native execution, while traditional whole-VM RR suffers from $8.97\times \sim 29.94\times$ slowdown. We validate that KRR is practical and has a broad recording scope by reproducing 17 bugs across different Linux versions, including 6 non-deterministic bugs and 5 high-risk CVEs; KRR was able to record and reproduce all but one non-deterministic bug.

1 Introduction

Modern kernels are particularly challenging to build correctly because they are huge and exceptionally complex. As a result of growing size and complexity, they are plagued with bugs. The Linux kernel, for example, has more than 30 million lines of source code and Syzkaller alone, a popular automated testing tool, found 3736 bugs in a 3-year span [55]. While many kernel bugs are routinely found through code reviewing and testing, many others end up finding their way to stable releases, leading to production failures in real-world deployments with drastic consequences for users [45–48, 51, 52, 65].

Record-replay (RR) techniques [40, 42, 49, 53, 56, 60, 64, 68] are particularly effective at helping developers diagnose non-deterministic and complex failures like those often found in kernels. At a high level, RR techniques rely on instrumenting a program (or VM) to record all input events that impact its execution into a trace such that, during a replay run, the same input can be fed to the program (or VM), reproducing the execution with 100% accuracy. This allows repeatable executions of the failure and the use of sophisticated offline analysis on the replayed execution, such as reverse debugging [43] and automated and expensive techniques [40]. Because of its accuracy, RR techniques are particularly useful for the most complex bugs. However, despite its appeal and decades of research on record-replay techniques, even the most well-developed record-replay frameworks, like Mozilla RR [56], tend to have very high overheads, particularly under multi-threaded or multi-core environments where the overhead factor exceeds the thread/core count [49, 56, 60].

This work presents KRR, an efficient and scalable record-replay framework that is specially designed to assist kernel developers in diagnosing kernel failures [32, 46, 47]. KRR aims to support modern data center workloads and multi-core VMs with significantly lower overheads than existing approaches. Like other approaches, KRR replays executions at the VM level; however, unlike existing approaches, KRR only records and replays the kernel execution. This approach departs from existing approaches in that the record and replay boundary is a slice of the software stack—the kernel—introducing significant technical challenges and an opportunity for major scalability and performance improvements.

The design of KRR relies on two key observations about modern data center workloads and the state of the art of RR techniques. First, modern workloads are increasingly concurrent; hence, supporting multi-core executions is critical even for emerging data center paradigms like serverless computing. Unfortunately, existing RR tools for x86 VMs have overhead factors that exceed the number of cores for gen-

eral workloads [42, 60]. For instance, recording a 2-core VM adds a 2.3-3.5× overhead [60]. Second, data center workloads are increasingly distributed and I/O intensive, a trend that has motivated the thriving field of kernel-bypassing techniques [33, 58, 59, 72]. Unfortunately, the higher the VM I/O, the more event data needs to be recorded by existing whole-VM RR approaches, increasing recording overheads. Hence, a practical RR tool should support well workloads that are concurrent and I/O intensive, but these workloads are exactly the less well supported by current RR techniques.

This work turns this problem on its head by precisely targeting and exploiting these workloads' properties – concurrency and I/O intensity. Since data centers increasingly use kernel-bypassing techniques to avoid kernel overheads, the bulk of the VM input is no longer kernel input. Hence, recording only the kernel input under these settings can reduce the input that needs to be recorded to replay the kernel when compared with a whole-VM replay. Second, under typical workloads, and even more so under kernel-bypassing workloads, cores are most often not executing kernel code; they are typically executing application code. Therefore, restricting the schedule recording to the kernel execution can significantly reduce the scalability impact on end-to-end user workloads. While combining multi-core VMs and kernel bypass is expected in demanding scenarios, our findings show that the KRR approach offers significant performance improvement even when only one property is present, underscoring its particularly broad utility.

Traditional whole-VM RR requires hypervisor modifications to trap all hardware-level inputs. In contrast, KRR narrows the record-replay boundary to the kernel, recording software inputs such as system calls and user data (e.g., via copy_from_user) alongside hardware inputs. Although this kernel-focused approach appears simple, it introduces fundamental challenges that require a novel split-recorder architecture. KRR divides recording responsibilities between the hypervisor recorder and the in-guest recorder. The in-guest recorder captures software inputs to the kernel and hardware inputs that bypass the hypervisor, while the hypervisor recorder handles remaining hardware events. This design addresses four key challenges: First, KRR must record more event classes than VM-RR, managing two recording interfaces rather than one. Second, it must coordinate cross-layer event recording to establish a total order without costly VM exits. Third, the kernel must serialize itself and record serialization order without hypervisor involvement to avoid VM exits. Fourth, since replay execution is a subset of recorded execution, injecting non-deterministic events requires a new mechanism beyond traditional instruction counts. All these challenges require a careful design of both the record and replay mechanisms.

We implemented KRR on KVM/QEMU and Linux, along with a comprehensive validation mechanism to check correctness through testing. We evaluate KRR against traditional whole-VM RR using real-world applications, including RocksDB and kernel compilation workloads. Our results show that KRR has high recording efficiency and scalability. For example, when recording 8-core VMs that run RocksDB and kernel compilation, KRR incurs only a slowdown of 1.52× ∼ 2.79×—significantly outperforming traditional whole-VM RR, which introduces slowdowns of 8.97× ∼ 29.94×. Furthermore, under kernel-bypass configurations, KRR achieves even lower overhead by allowing guest user-space applications to communicate directly with the hardware, incurring no recording cost on these data paths. Thus, in a 4-core VM running the Redis benchmark, KRR incurs at most 1.05× runtime overhead on throughput, whereas in a 2-core VM running RocksDB, the slowdown ranges from 1.17× to 1.26×. Single-core workloads with kernel-bypass similarly benefit from KRR 's efficient recording design.

To evaluate the effectiveness of recording real-world bugs, we applied KRR to reproduce 17 bugs across different Linux versions, including 6 non-deterministic bugs and 5 high-risk CVEs. KRR was able to record and reproduce all but one non-deterministic bug, showing that KRR is effective at recording and replaying complex and serious real-world kernel bugs, including many non-deterministic bugs.

2 Sliced Record Replay

Record-replay techniques record executions and replay them at a later time, allowing developers to conduct postmortem analysis of faulty executions. Record-replay approaches have been applied at the level of applications and virtual machines. In practice, this approach records all nondeterministic inputs into a trace log and feeds them into another execution, reproducing the control and data path of the original execution, while giving an opportunity for developers to retroactively apply powerful analyses and debugging techniques. These analyses are instrumental to help developers diagnose the most complex failures, such as failures where the root cause is distant from the failure point [37, 40, 73].

Key observations. This work makes two key observations. First, although the kernel is often the most complex system running on a machine, it is not the one that is most often running on a given core. Second, when data-intensive applications use kernel-bypassing, input provided to the kernel can be significantly smaller than the input provided to the machine. Combined, these observations make us question the common approach to recording kernel executions, which relies on recording the non-deterministic inputs of the whole VM. Instead of recording the entire VM execution, we postulate that recording only the kernel execution, i.e., a slice of the stack, may allow us to achieve much better recording performance. However, as explained next, this represents a significant change to record-replay techniques, which currently assume a single interface for the record and replay boundary

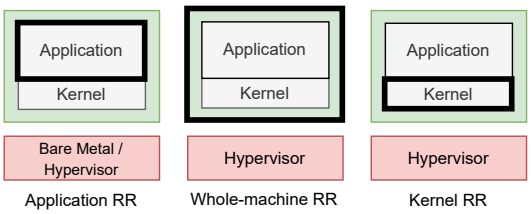


Figure 1: Traditional RR approaches and Kernel RR approach. Record-replay boundary is highlighted in bold.

(e.g., the system call or machine interface), posing unique challenges to the tool design.

2.1 Goals and Assumptions

We aim to build a record-replay framework that (a) records kernel executions, (b) ensures matching, functionally-accurate kernel replays, and (c) achieves high *recording* efficiency. Our goal is to drastically reduce the recording overhead of traditional record-replay frameworks, especially in multi-core environments, to enable recording all executions of large-scale workloads.

Non-goals. Replaying efficiency beyond ensuring sufficient speed for practical debugging is not a goal of this work. Every replayed execution has to be recorded, but often only some recorded executions – those that happen to trigger failures – are replayed. Replaying application executions is not our focus since we aim to help kernel developers find and diagnose kernel bugs. We assume that the kernel is designed to work correctly regardless of the input provided by applications, hence determining why applications provide certain input that leads to kernel failures is out of the scope of this work.

Assumptions. Since we aim to help kernel developers fix kernel bugs, we assume that they can modify the kernel. Furthermore, we assume that the kernel executions run on a virtualized environment, which is typical in the cloud and datacenters, but that developers or data center providers can modify the hypervisor, which is a standard assumption for VM-based techniques. Our design is architecture-agnostic, although our implementation and discussion targets x86 due to its wide adoption in data centers.

2.2 Efficient and Scalable Kernel RR

Accurately replaying executions requires recording all non-deterministic inputs of a given execution. In practice, this means that all the explicit inputs (i.e., inputs provided by users) and implicit inputs (e.g., program schedules) need to be recorded such that they can later be fed into the program, ensuring the same execution behavior during replay.

To record and replay kernel executions, the state-of-the-art approach records the whole system execution (Whole-machine RR in Figure 1). In practice, this is done with a modified hypervisor that records all the VM (implicit and explicit) inputs. By running the target kernel inside a VM, the record and replay boundary becomes the entire VM. In contrast to the prior approach, KRR restricts the *recording and replaying boundary* to the guest kernel from the entire VM (Kernel RR in Figure 1), therefore avoiding the unnecessary cost of recording the guest applications.

At first glance, it may appear that by recording a smaller layer of the software stack (i.e., the kernel layer), the recording overheads become smaller. But the size of the layer does not directly determine the costs. In fact, reducing the RR boundary to the kernel could increase costs because a sliced recording that only targets the kernel would require recording (a) the hardware input provided to the kernel (e.g., data read from the disk or network and interrupt timings) and (b) the input provided by applications to the kernel (e.g., system calls invoked by the applications). Whereas recording the VM execution would only require recording the former source of (explicit) inputs. Despite this additional cost to the sliced RR approach, there are two factors that make KRR not just viable but significantly more efficient and scalable than whole-machine RR.

Adoption of kernel-bypass workloads. Because datacenter workloads increasingly use kernel bypassing for higher performance, the input provided to the kernel can actually be smaller than the input to the machine. Hence, under kernel-bypassing workloads, restricting RR to the kernel boundary can reduce the number of recorded events, thereby increasing recording efficiency.

High cost of schedule recording. The typical Achilles heel of RR techniques is parallelism. The few RR systems that support multi-core machines (or multi-thread applications) on commodity hardware primarily do so by serializing the execution of multiple cores (or threads) [56], which cancels the parallelism benefits, or recording memory access orderings, which has comparable or higher costs [42, 60]. Thus, the state-of-the-art RR techniques for x86 have overheads that exceed the number of cores for general workloads (e.g., 2.3-3.5× overhead for a 2-core VM [60]), canceling the performance benefits of using multi-core machines. Sliced RR that selectively targets the kernel can drastically mitigate this cost, at the end-to-end level, by only serializing the execution of the kernel while allowing full application parallelism. Since in many workloads, the vast majority of the CPU time is spent in user mode, sliced RR can scale kernel recording to several cores on commodity hardware with low CPU overhead. It is worth noting that despite the use of serialization to handle parallelism, as discussed later (§3.3), fine-grained serialization is effective even at recording failures that trigger non-deterministic, including important classes of concurrency

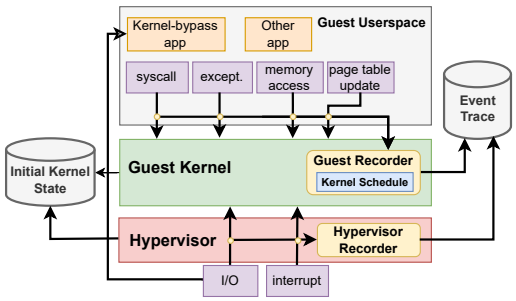


Figure 2: KRR Split Recorder Architecture.

bugs.

3 KRR Design

KRR's unique recording boundary imposes two major design challenges. First, while existing record-replay systems focus on either the user-space (application RR) or hardware interface (whole-machine RR), KRR must simultaneously record both interfaces, as illustrated in Figure 1. This dual-interface recording requirement creates significant architectural challenges. Traditional approaches can record from a higher-privilege level—using the kernel to record the application, or the hypervisor to record the VM. However, neither interface provides a good vantage point to effectively record both interfaces given the limited visibility from each interface. The second challenge lies in the recording efficiency. Because the kernel frequently interacts with user space and hardware and often with fine granularity, intercepting communications over these two interfaces can be expensive, particularly when recording from the hypervisor. Naively recording even just some event types could cause frequent VM exits or require expensive introspection mechanisms that we cannot afford.

To address these challenges, KRR employs a split-recorder design (Figure 2) that uses an in-guest recorder and an in-hypervisor recorder to collaboratively record the kernel execution. This design enables KRR to record both interfaces effectively and with low overhead. Together, the two recorders record all non-deterministic events that impact the kernel execution and output a recorded trace of events in total order, allowing KRR to accurately reproduce the slice of the kernel execution during replay by injecting the recorded events one by one. Specifically, KRR records three types of events: (1) all inputs provided to the kernel (§3.1 and §3.2), (2) the kernel schedule and other non-determinism (§3.3), and (3) the initial state of the kernel (§3.4).

3.1 Kernel Input from User-space

KRR implements an in-guest recorder that monitors all input, both directly and indirectly, provided to the kernel by applications. The guest recorder runs in the guest kernel and intercepts the kernel's accesses to software input. Fortunately, the kernel interface is generally very well defined for maintainability and security reasons, making it practical to determine the source code locations where the kernel receives input. The in-guest recorder handles several sources of software input.

System Calls. Applications invoke a system call by requesting the index of the desired call and passing the respective arguments. Thus, correctly replaying the system call requires recording the requested call index and the arguments. To do so, the guest recorder monitors the common entry point of system call handlers. When a system call request is received, the guest recorder accesses the CPU registers to get the call index and parameters.

User-Memory Accesses. The kernel can read user-space memory to fetch application structures, buffers, and for other reasons, which therefore constitutes another type of input event for the guest recorder to monitor. For instance, to fulfill a `write()` system call, the kernel needs to read the user-space memory to fetch the contents to write. In abstract, monitoring user-memory accesses is challenging because the kernel can make the read accesses at arbitrary points during its execution. An interesting observation that makes our solution particularly feasible is that modern kernels treat user-memory reads as security-sensitive operations due to their potential as attack vectors. As a result, such reads are performed through a small set of internal APIs that have been thoroughly reviewed and hardened with security features such as the x86 SMAP extension. This design property allows KRR to reliably record user-memory reads by instrumenting well-defined API functions such as `copy_from_user` and `get_user` [30]. When the kernel reads data from user-space through the instrumented functions, KRR inspects the corresponding destination kernel-space memory to record the copied content. Importantly, KRR does not access the user-space memory directly for recording, thereby avoiding consistency issues such as double reads (e.g., TOCTOU [29]).

Additionally, the kernel's execution could be affected by inputs from memory that is shared between the kernel and user-space. A typical example is `io_uring` [15], which uses shared circular queues for asynchronous I/O operations. With `io_uring`, the kernel directly accesses the submission queue in the shared memory region, which the user-space can access and modify at the same time. KRR records this type of kernel input by instrumenting kernel reads to both the submission queue and its entries in the guest recorder, ensuring that user inputs via this shared memory are captured deterministically. This approach is also applied to other mechanisms, such as inter-process communication (`process_vm_readv`), and can be extended to other shared-memory inputs that affect kernel behavior.

Page Table Updates. In addition to the explicit inputs described above, user-space execution can implicitly influence

kernel behavior through changes to the page table state. Specifically, when applications access memory, these accesses may update the `accessed` and `dirty` bits in the corresponding page table entries. These updates, in turn, can affect the kernel memory management, as the kernel relies on those bits to make decisions such as page replacement. To record the page table entry update, KRR instruments kernel accesses to the entries and records their exact values. This approach is generic and can record other forms of implicit inputs if needed. In practice, our comprehensive validation and evaluation (§4.1, §5) suggest that page table state changes are the only typical implicit input provided by processes, and recording these changes is sufficient to ensure accurate replay.

3.2 Kernel Input from Hardware

Hardware inputs (interrupts, I/O read, and device DMA) are primarily recorded by the hypervisor recorder, while non-deterministic instructions and exceptions are recorded by the guest to avoid introducing VM exits.

Interrupts and Exceptions. The hypervisor recorder intercepts every interrupt to record the interrupt type (i.e., vector number), timing (§3.2.1), instruction pointer (RIP), and all general-purpose registers that may be accessed during interrupt handling. For example, RCX register is accessed to identify the position within repeating string instructions [42, 60], and RIP is accessed to handle restartable sequences [18].

Because KRR only records the kernel slice of the execution, exception handling, such as those caused by division by zero from user-space, are no longer deterministic events in every situation. In particular, exceptions triggered by applications need to be recorded and injected during replay by KRR. For each exception, KRR records the exception number, all general-purpose registers, and any additional machine state required by each specific exception handler. For instance, when recording page faults on x86_64, the guest recorder additionally records the faulting memory address and error code stored in the stack memory.

I/O Operations. KRR records the data read by the kernel from hardware devices through port I/O (i.e., `IN` and `OUT` instructions), memory-mapped I/O (MMIO), and DMA accesses. Since they are handled by the host hypervisor, KRR uses the hypervisor recorder to monitor these events. When an I/O read request is intercepted, the hypervisor records it only if initiated by the kernel. By excluding I/O requests from guest user-space applications, KRR significantly improves recording efficiency for kernel-bypass workloads (§5.2).

Other Non-deterministic Instructions. Because some instructions produce non-deterministic results, KRR needs to record their results to ensure correct replay. On x86, this includes instructions such as `RDTSC`, `RDTSCP`, `RDSEED`, and `RDRAND`. While a common approach is to trap and record them from the host hypervisor [60, 64], KRR instead records them with the guest recorder to avoid expensive VM exits. This shows another advantage of choosing the kernel as the record boundary – it would be impractical to ensure that these instructions are consistently recorded for every application in a VM-RR approach by modifying all applications, since there is no intra-VM trap mechanism for these instructions in x86. However, it is practical to manually modify a single system (the kernel, in our case) to intercept them.

3.2.1 Asynchronous Event Timing

Unlike other hardware inputs, interrupts and DMA accesses are asynchronous events. Thus, their timing has to be recorded precisely so that KRR can replay them at the exact same point observed during recording. To achieve this, KRR leverages x86 kernel-mode-only hardware counters to track the number of executed kernel instructions during recording. When the interrupt or DMA event happens, KRR records the counter value as its timing signature, enabling accurate replay by injecting the event at the same instruction count.

To prevent conflicts with the guest kernel over hardware performance counter usage, KRR takes advantage of the multiple counters available on modern processors (e.g., Intel processors typically provide four programmable counters per core [13]). KRR reserves one counter for its exclusive use, leaving the remaining counters accessible to the guest kernel. This is achieved by intercepting the `CPUID` instruction's output to mask the reserved counter, thereby making it invisible to the guest.

3.3 Kernel Schedule

A common approach to ensure accurate record and replay on concurrent software is to serialize the multi-thread execution—allowing only one thread to run at a time—thereby resolving data races and other schedule-non-determinism during recording. However, while this approach guarantees correct replay, it can impose substantial performance overhead by preventing parallelism. To address this limitation, KRR takes advantage of its reduced recording boundary. Since KRR records the kernel execution exclusively, it only has to serialize the concurrent kernel execution, while still allowing user-space threads to run in parallel.

In addition, unlike other approaches that serialize at the hypervisor-level [44], KRR serializes the execution from the guest kernel, avoiding expensive VM exits on every kernel-user space transition. KRR uses a special-purpose spinlock from the guest kernel, ensuring that only one vCPU core can execute in kernel mode. Although this approach may seem simple, not using the hypervisor to control the schedule requires a careful design, as discussed next.

Replay-Coherent (RC) Spinlock. While the serialization spinlock is integrated into the guest kernel, its execution

should not be replicated during replay since it is, conceptually, a component of the recorder and it would slow replay. Even more importantly, a normal spinlock is itself non-deterministic, meaning that the spin count (instructions executed by the spinlock) and its outcome depend on the schedule. If left unhandled, the outcome of a normal spinlock would not match across record and replay. Furthermore, even if the outcome of the spin lock is recorded, the spin count difference across runs would make the instruction count inconsistent across runs, preventing KRR from using the instruction count to decide when to inject interrupts and other asynchronous events that are timing-dependent (§3.2). After exploring several approaches to address this problem, including relying on expensive VM exits, we concluded that a simple modification to the spinlock is sufficient to solve this problem. KRR introduces a new spinlock design, the RC spinlock, that counts the number of cycles (instructions executed) before it is acquired as well as the acquisition ordering, which KRR records as events. This allows KRR to enforce the same ordering and adjust the instruction count to match the recorded values during reply, thereby reproducing the exact multi-threaded kernel execution

Deadlock Prevention. KRR manages kernel execution serialization through its RC spinlock. The vCPU acquires this lock when (a) entering kernel mode (through system calls, interrupts, or exceptions) or (b) waking from an idle state, and releases it when (a) returning to user-space or (b) entering the idle state.

However, this basic locking scheme can lead to deadlocks. For example, when a thread holding the RC spinlock waits for an internal kernel lock that is owned by another thread, which in turn is waiting for the RC spinlock. To prevent such deadlocks, KRR adds another acquisition/release point: the kernel releases the RC spinlock before acquiring certain internal locks (including the CSD lock [17] and spinlock) and reacquires it afterward.

When multiple vCPUs wait for the same internal lock, KRR applies a similar strategy as used for the RC spinlock - recording the acquisition ordering to ensure correct replay. Since the waiting time at internal locks is non-deterministic, KRR uses hypercalls to synchronize instruction counts between record and replay runs. While hypercalls incur VM exits, these synchronization points are not common during normal kernel execution, resulting in a low performance impact.

Bug Recording Scope. Like other practical record-replay tools, KRR affects executions through subtle changes to the timing of events due to instrumentation overheads or the serialization mechanism, making some failures potentially harder or easier to trigger. Despite the use of serialization, KRR can record concurrency bugs, as our experience shows. This is possible because KRR allows concurrency, although at a coarser grain than native executions, through interrupts and context switches. To empirically assess KRR's ability to reproduce real-world bugs, we evaluated it on a diverse set of Linux kernel bugs (details in §5.3). The results show that KRR successfully reproduces 16 out of 17 randomly-sampled bugs, encompassing 5 out of 6 non-deterministic bugs and all five tested kernel CVEs. This validates its effectiveness for a broad spectrum of real-world kernel problems, including several concurrency-related bugs. §6 discusses further the scope of the recorded executions.

3.4 Initial Kernel State

KRR allows users to start recording the guest kernel execution at arbitrary points during the VM lifetime. This selective recording capability is particularly useful in performance-critical cases where only specific execution periods are recorded due to lower overhead tolerance (e.g., during intensive workloads).

Once recording starts, KRR takes a memory snapshot of the VM. Unlike traditional VM-RR, KRR requires only a kernel memory snapshot for replay, presenting a significant optimization potential. Snapshotting only kernel memory, instead of the entire VM's memory, can substantially improve snapshot efficiency and reduce storage costs, particularly for VMs with large amounts of user memory. This snapshot is then used in the replay to ensure that the event trace is applied to a kernel execution that starts from the same initial state (§3.5). Currently, KRR uses synchronous snapshotting where the guest's execution is paused during snapshot creation to ensure state consistency. However, future work could adopt techniques, such as asynchronous snapshotting [1] and efficient fork, like on-demand fork [74] and others [69], to further reduce the performance impact of snapshotting operations.

3.5 Replayer

During replay, KRR reconstructs kernel execution through a carefully orchestrated process. KRR first loads the VM snapshot and follows the recorded event trace to recover the original execution environment. The system employs several techniques to ensure identical execution: vCPU thread scheduling based on logged execution order, memory state recovery using VM breakpoints to intercept memory accesses to user-space memory, hardware input replication during I/O operations, and precise timing control through instruction counting to inject asynchronous events at the exact points they occurred during recording.

Replay Debugging. Since performance is less critical during replay, we implemented our KRR prototype replayer using the QEMU emulator, which has the advantage of simplifying introspection and providing a convenient kernel debugging environment through GDB integration. Our implementation allows developers to use powerful debugging features including breakpoints, watchpoints, and automation scripts [16] to analyze the kernel behavior during replay. Moreover, other

binary analysis techniques [37, 40, 61] can be integrated into KRR, allowing developers to track memory accesses, monitor register changes, and analyze execution paths at instruction-level granularity.

Reverse Debugging. Reverse debugging [11] enables developers to navigate backward through a program's execution history to examine how a particular state was reached. This capability is invaluable for diagnosing complex bugs whose root causes are difficult to pinpoint with forward-stepping debuggers alone.

To implement reverse debugging, KRR augments its standard kernel replay by also periodically taking VM snapshots, each tagged with the instruction count. To start reverse debugging, e.g., a backwards instruction step, KRR performs a two-stage process. First, it identifies and loads the last VM snapshot that was captured before the target backward point. Second, it replays the execution forward from the snapshot, using the recorded event trace, until the exact desired execution point is reached. This approach is conceptually similar to QEMU's native reverse debugging [20].

However, robust reverse debugging for multi-core VMs presents challenges not addressed by existing approaches [21, 40]. In multi-core systems, each vCPU progresses independently, making a single instruction counter insufficient to uniquely define the system state for consistent reverse execution. To overcome this, KRR introduces a *multi-core execution coordinate*: a vector of per-vCPU instruction counts captured during replay. Both snapshots and past breakpoints are tagged with these coordinates during replay. During a rollback to the last breakpoint, KRR loads the closest prior snapshot and replays forward until the target coordinate is reached, restoring both VM state and replay progress to ensure consistency.

4 Implementation

We implement KRR based on Linux-KVM (5.17.5) and QEMU (7.0.0). We develop the guest recorder on Linux 6.1.0 and then port it to different kernel versions, ranging from 5.10 to 6.1 (§5.3). The modifications to KVM, QEMU, and the guest Linux are around 1.2k, 4.5k, and 1.2k lines of C code, respectively.

Guest Recorder. KRR implements a kernel recording library with 16 APIs that record system calls, exceptions, user-memory access, page table accesses, non-deterministic instructions, and RC spinlock operations. The library, implemented in about 1.2K LoC C code, is used to instrument 37 kernel source files for recording these events. The recording library also facilitates compatibility and portability across kernel versions. When using KRR to reproduce known kernel bugs (§5.3), we are able to support 13 unique kernel versions ranging from 5.10 to 6.1. Supporting a new kernel version typically requires less than 30 minutes to apply our patch files and resolve code conflicts.

Hypervisor Recorder. We implement the hypervisor recorder in KVM/QEMU to capture hardware-related events. The implementation consists of modifications to KVM (0.8K LoC) and QEMU (1.2K LoC), providing 5 instrumentation interfaces for recording. In total, we modify 7 files across the KVM/QEMU codebase. We instrument KVM's interrupt injection function to record interrupt information and extend PIO & MMIO emulation functions to record I/O read results.

Hypervisor recorder records data from both the emulated disk and network interfaces, with each requiring different handling strategies due to their distinct access patterns. For disk I/O, KRR records data written to device DMA memory regions and pairs each DMA buffer chunk with its triggering disk I/O instruction (e.g., IDE DMA read command). This pairing, along with the timing information, enables precise injection of recorded DMA data during replay when the kernel executes the corresponding I/O instruction. Network I/O presents a unique challenge due to Linux's NAPI mechanism [19]. NAPI allows concurrent access to the network ring buffer by both the kernel and the device, creating potential data races. KRR addresses this by trapping the vCPU in kernel mode before the device writes to the network ring buffer, similar to VMware FT's network I/O synchronization approach [63]. During this trap, KRR records both the data written to the ring buffer and the current instruction count, enabling accurate replay of network inputs. For kernel-bypass devices, KRR provides a QEMU parameter that allows users to specify which device inputs should be ignored during recording.

Atomic and Ordered Event Recording. Because KRR uses both the guest kernel and hypervisor recorder to record input events, it is critical that the event trace is updated atomically and sorted in the correct order (i.e., by occurrence time). To guarantee atomicity, KRR grants exclusive access to the event trace only to the RC spinlock owner vCPU. When one recorder is updating the event trace, KRR disables the interrupt for the guest and pauses the corresponding vCPU core on which the event happens until the event is recorded into the trace. By doing so, KRR ensures that only one recorder can update the event trace at a time.

While KRR serializes kernel execution, interrupts may still occur concurrently on multiple vCPUs that are in user mode, making it challenging to order these interrupt events correctly. KRR's recorded ordering of RC spinlock acquisitions (§3.3) ensures these interrupts are properly ordered according to the serialized execution.

4.1 Validation

We run the test suite from Linux Test Project (LTP) to test the correctness of our prototype. In the recorded execution, at every *N-th* instruction, we log the instruction pointer (RIP), the instruction counter (i.e., number of executed kernel instructions) and all `x86_64` registers, including control, general-

purpose, and segment registers. During the replay, we assert that the replayed execution shows consistent states at every *N-th* instruction. We use the value 1K for *N* by default and increase it to 32K if too frequent logging causes test timeouts during recording. Overall, KRR can successfully replay the kernel execution of 8,156 LTP tests, demonstrating its high recording accuracy.

5 Evaluation

This section evaluates KRR along the following questions:

RQ1: Can KRR efficiently record multi-core workloads? (§5.1)

RQ2: What is the recording performance of KRR on kernel-bypass workloads? (§5.2)

RQ3: Can KRR record/replay complex kernel bugs? (§5.3)

RQ4: What is the storage cost of KRR? (§5.4)

RQ5: What is the replay performance of KRR? (§5.5)

Experimental setup. We evaluate the record/replay performance of KRR on the CloudLab platform [3] using instances of machine c6420, which is equipped with two 16-core Intel Xeon Gold 6142 CPUs (hyper-threading and cstate disabled), 384 GB RAM, and a dual-port Intel X710 10GbE NIC for performance. In addition, we evaluate the effectiveness of KRR in reproducing kernel bugs on a separate machine with a 2-core Intel i7-7660U CPU and 16GB RAM. We run the modified Linux kernel 5.17.5 on the host, and 6.1.0 on the guest in the default recording performance evaluation. For the bug reproduction in §5.3, we run 13 unique guest Linux kernel versions ranging from 5.10 to 6.1.

VM-RR. Our evaluation compares the recording performance between KRR and the whole-machine RR approach(Figure 1). A direct comparison with prior whole-machine RR implementations is unfeasible for several reasons. First, some implementations are proprietary or have been deprecated [42, 64]; thus, they are inaccessible to us. Second, despite our considerable adaptation efforts, the old codebase of some systems caused compatibility issues on modern hardware [60]. Third, the other alternatives lack multi-core VM support or critical hardware-assisted virtualization capabilities, such as KVM, which are central to our work [21, 40]. Thus, to enable a fair and relevant comparison, we implemented VM-RR, a whole-machine RR prototype using the same version of QEMU and KVM as KRR, which is the fastest implementation we have access to (§4).

VM-RR differs from KRR in three aspects. First, it uses the traditional, wider recording boundary: VM-RR captures all hardware inputs to the VM, including non-deterministic events (e.g., interrupts), the results of non-deterministic instructions, and data from device I/O reads. KRR, in contrast, must also record inputs from the guest user-space to the kernel (§3.1) but does not record VM input that bypasses the kernel. Second is the handling of non-deterministic instructions (e.g., `RDTSC`): VM-RR relies on hypervisor traps to capture their output, whereas KRR uses its in-guest recorder to log these values, avoiding expensive VM exits (§3.2). Third, VM-RR enforces determinism through full-system serialization, allowing only one vCPU to execute at any time. vCPUs are scheduled via FIFO, each running for up to a 50K instruction time slice. While this serialization incurs overhead, like it does in KRR, it is a standard technique in prior deterministic systems [44, 46, 56]. Importantly, our evaluation results (§5.1.2) demonstrate that VM-RR's performance overhead is comparable to that of existing deterministic record-replay systems on multi-core workloads, affirming its validity as a representative baseline.

Methodology. To understand the recording overhead of KRR and VM-RR, we measure the system performance both with and without recording enabled, using the native execution (no recording) as the baseline. We conduct multiple trials of each experiment and report the average result. To ensure consistent measurements across trials, we clear the host page cache, pin every vCPU thread to a dedicated physical core, and boot the VM from scratch between trials. Our evaluation graphs present both the absolute measurement values and the relative slowdown caused by recording.

5.1 Recording multi-core workloads

This section evaluates the recording overhead of KRR on modern applications, with a focus on scalability.

5.1.1 RocksDB

First, we compare KRR and VM-RR on a modern key-value store, RocksDB [41], using its built-in benchmark suite [2]. We run multi-threaded workloads, aligning the number of threads with the available vCPU cores, to understand the recording overhead for concurrent RocksDB operations.

Throughput. Figure 3 illustrates the substantial performance disparities between VM-RR and KRR when recording multi-core RocksDB workloads. While all workloads demonstrate strong multi-core scaling under native execution, their throughput under VM-RR decreases greatly as the number of cores increases. Notably, with VM-RR, RocksDB performs worse on multi-core VMs than on single-core VMs across all workloads. In contrast, KRR introduces considerably lower overhead. On single-core VMs, KRR is expected to be slower than VM-RR because it needs to record userspace inputs while VM-RR does not (§3.1). However, contrary to this expectation, VM-RR is observed to be slower than KRR on certain workloads, despite avoiding userspace input recording. Our analysis shows that this is because VM-RR incurs significant overhead when handling `RDTSC` instructions, as

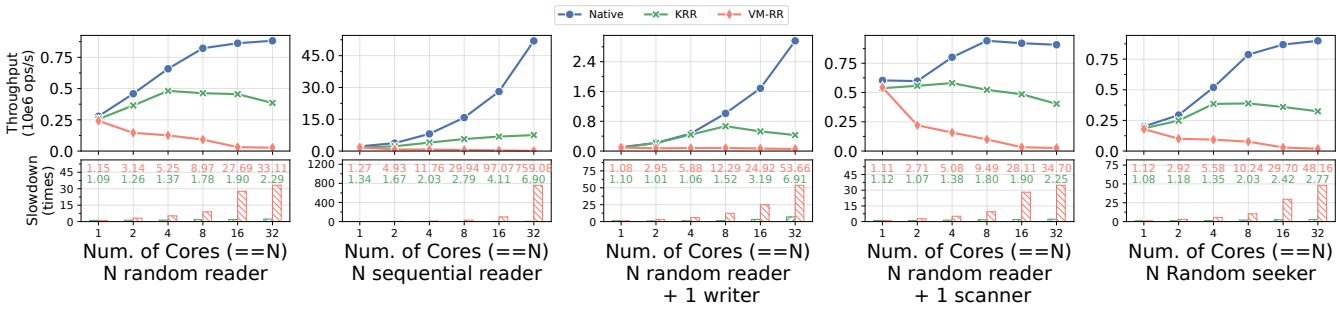


Figure 3: RocksDB benchmark throughput. Slowdown is the ratio between the native and recording throughput.

each instance necessitates a costly VM exit. The performance impact becomes substantial with higher instruction frequencies. For example, the *random reader* and *random seeker* benchmarks incur over 18k and 14k `RDTSC` instructions per second, respectively, while other benchmarks incur less than 7k. On 2-core VMs, KRR incurs a slowdown ranging from 1.01× to 1.67×, and on 4-core VMs, the slowdown ranges from 1.06× to 2.03×. These slowdowns are markedly lower than those observed with VM-RR, which range from 2.71× to 4.93× on 2 cores and 5.08× to 11.76× on 4 cores. Due to this moderate overhead, KRR enables most workloads to achieve increased throughput as the number of cores increases from 1 to 4. However, beyond 8 cores, RocksDB's throughput, under KRR, begins to decline. Our investigation reveals that the increasing lock contention on large VMs is the primary performance bottleneck.

Latency. The latency measurements mirror the throughput patterns described above. From 2-core to 8-core VMs, the operation latency under VM-RR, is slower than the native by 2.73× ∼ 29.99×, while for KRR, the latency is only increased by 1.01× ∼ 2.80×.

5.1.2 Kernel Compilation

We next evaluate KRR and VM-RR using a code compilation workload on VMs with 1, 2, 4, 8, 16, and 32 vCPU cores and 8GB of memory. Within the VM, we compile the Linux kernel 6.5.1 using the command `make -j $(#cores + 1)`. The number of worker threads is set to one more than the number of cores to better saturate the CPU [42, 60].

Figure 4 presents the compilation time with and without recording and the slowdown caused by KRR and VM-RR. On a single-core VM, both KRR and VM-RR introduce moderate slowdowns of 1.15× and 1.11×, respectively. The slightly higher slowdown observed with KRR is because KRR needs to additionally record guest user-space inputs to the kernel, whereas VM-RR records only hardware inputs. However, the scaling behaviors of KRR and VM-RR diverge significantly when recording multi-core VMs, where concurrency-induced non-determinism must be recorded. With KRR, kernel compilation scales effectively up to 8 cores, resulting in only 22%

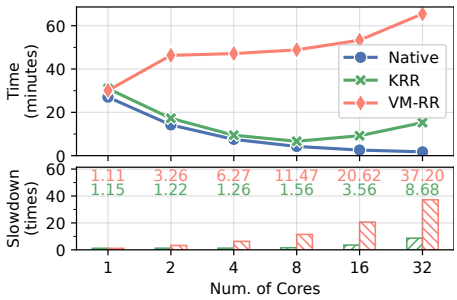


Figure 4: Linux Kernel build time. Slowdown is the ratio between the recorded build time and the native build time.

and 56% slower than the native on 2-core and 8-core VMs. In contrast, with VM-RR, kernel compilation does not scale with more CPU cores. The performance under VM-RR is 226% and 1047% slower than the native on 2-core and 8-core VMs. Similar to the RocksDB workload, kernel compilation, under VM-RR, performs worse on multi-core VMs than on single-core VMs. This finding aligns with prior work on whole-machine RR [42, 60], which reported that kernel build performance on 2-core VMs is slower than on single-core VMs.

For VMs with 16 and 32 cores, KRR's kernel build no longer benefits from more CPU resources. The performance downgrade, similar to the results observed with RocksDB, stems from the increased lock contention between vCPUs as the VM size grows.

5.2 Recording Kernel-bypassing Workloads

The difference in recording boundaries allows KRR to achieve significantly lower overhead when recording kernel-bypass workloads. The following section quantifies the recording efficiency of KRR on three modern kernel-bypass applications: RocksDB (§5.2.1), Redis (§5.2.2), and Nginx (§5.2.3).

5.2.1 RocksDB with SPDK

We run RocksDB with SPDK to directly access the disk from the user-space. In this experiment, we run several single-threaded RocksDB workloads in a 2-core VM with 8GB of memory and an emulated NVMe device. Following the SPDK best practices [27], we pin the RocksDB worker thread to one vCPU core and the SPDK polling thread to another, which continuously monitors the NVMe device state and handles I/O requests from the worker. The NVMe device is emulated on a standard SATA disk with our modified QEMU (§4) that implements NVMe data recording, which is necessary for VM-RR and KRR—under the non-kernel-bypass mode—but is not supported by existing NVMe devices.

Throughput. Figure 5 compares the throughput of different RocksDB workloads with and without SPDK. The results show that several workloads benefit from bypassing the kernel for I/O handling. For instance, in the *1 sequential writer* workload, RocksDB-SPDK achieves 3.27× higher throughput compared to RocksDB without SPDK. However, when VM-RR is used for recording, RocksDB-SPDK experiences significant performance degradation. Specifically, the throughput decreases by 29.37× in the *1 random appender* workload and 64.51× in *1 sequential deletion*. These slowdowns are substantially greater than those observed with standard RocksDB using kernel I/O, which exhibits slowdowns ranging from 1.67× to 2.30×. The severe performance degradation results from the whole-system serialization approach in whole-machine RR: when VM-RR blocks the CPU core running the SPDK polling thread but allows the core running the worker thread to execute, the worker thread stalls on pending I/O requests despite having CPU time. In contrast, the RocksDB worker can progress normally when using kernel I/O.

Conversely, KRR achieves reasonable recording overhead with RocksDB-SPDK by allowing user-mode polling threads to run alongside workers. For write workloads, KRR's overhead on RocksDB-SPDK is 10% $\sim$ 22.7% lower than on traditional RocksDB, primarily because SPDK drastically reduces both system calls (by 77% $\sim$ 94%) and userspace-to-kernel data copies (by 94% $\sim$ 99%). The *Sequential reader* benchmark also shows substantially lower slowdown under SPDK, as KRR does not need to record the substantial DMA data transfers that dominate this workload's intensive disk I/O operations.

Latency. When using kernel I/O, KRR consistently outperforms VM-RR with a much lower impact on operation latency. KRR's slowdown is within 1.14× $\sim$ 1.54× across all workloads, while VM-RR's slowdown is between 1.67× $\sim$ 2.33× For RocksDB with SPDK, VM-RR slows down its latency by 29.36× $\sim$ 64.51× compared to native execution. In contrast, KRR marginally affects the latency of RocksDB-SPDK, with slowdowns ranging from 1.17× to 1.27×.

5.2.2 Redis with DPDK

Next, we evaluate KRR on Redis with kernel-bypass. The evaluation utilizes two inter-connected c6420 machines on CloudLab: one for hosting the Redis server inside the VM, and another for running the Redis benchmark [24]. The VM is configured with 4 vCPU cores, 8GB memory, and has direct access (i.e., passthrough) to the dual-port Intel X710 10GbE NIC. Within the VM, we deploy the Redis implementation from the f-stack kernel-bypass framework [10], which leverages DPDK [9] for direct network card access. The client machine runs the Redis GET and SET benchmarks separately with 5 million requests and different numbers of concurrent client threads $\in \{1,2,4,8,16,32,64\}$. Due to the lack of data recording support in the physical NIC under this setup, VM-RR is unable to accurately record all hardware inputs to the VM. Thus, it is excluded from this evaluation.

Figure 6 shows that KRR can efficiently and scalably record the guest kernel under the Redis-DPDK workload, Across all numbers of concurrent clients, KRR only reduces the GET throughput by 0.26% and the SET throughput by 1.14% on average. Additionally, the P99 latency measurement exhibits slowdowns ranging from -5.19% to 11.27%. The minimal recording overhead is because the guest kernel receives significantly fewer inputs under Redis-DPDK. By delegating the heavyweight data path to the user-space, the guest kernel is primarily responsible for handling the I/O control path, thereby reducing the number of events that need to be recorded.

5.2.3 Nginx with DPDK

Using a similar setup to the Redis-DPDK experiment, we evaluate KRR's recording performance on Nginx with DPDK kernel-bypass (version 1.25.2, implemented in f-stack). On the server side, we use VMs with different numbers of vCPUs $\in 1,2,4,8,16,32$ and run Nginx with the matching number of worker processes. On the client side, we run the benchmark tool wrk [31] to request files of size $\in 1KB, 4KB, 16KB, 64KB$ over 1024 connections (32 connections per client thread) for a duration of 10 seconds.

Figure 7 shows that the recording overhead of KRR varies significantly with the requested file size. For small files (1KB and 4KB), KRR incurs a relatively high slowdown exceeding 46%, and this overhead becomes increasingly severe as the number of cores increases. However, for larger files (16KB and 64KB), the overhead of KRR becomes almost negligible, reducing to merely 2% for 16KB files and 5% for 64KB files across all VM sizes. Our analysis reveals that this discrepancy is due to the unique performance bottlenecks in serving files of different sizes. For small files, the bulk of the request processing time is spent in the kernel reading the file. Since KRR instruments these kernel operations, the recording overhead causes a significant slowdown in the end-to-end throughput. In contrast, when serving larger files, the primary performance bottleneck shifts to the network transfer. Under

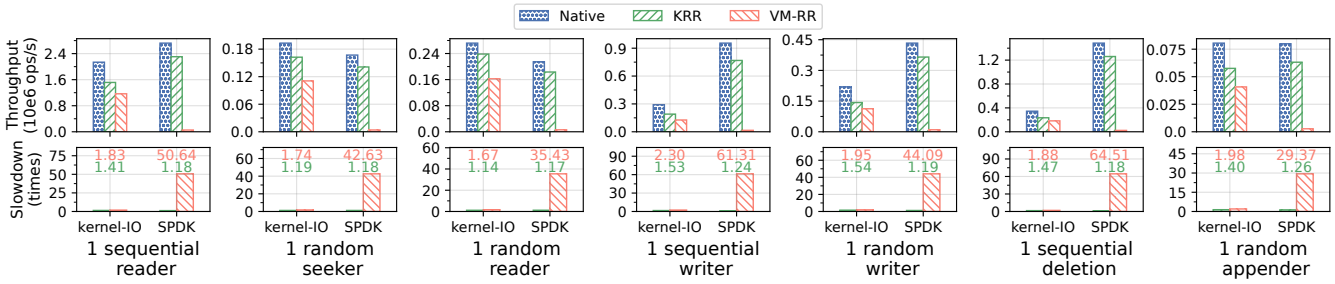


Figure 5: Throughput of RocksDB w/ and w/o SPDK on single-threaded workloads. Slowdown is the ratio between the native and recording throughput.

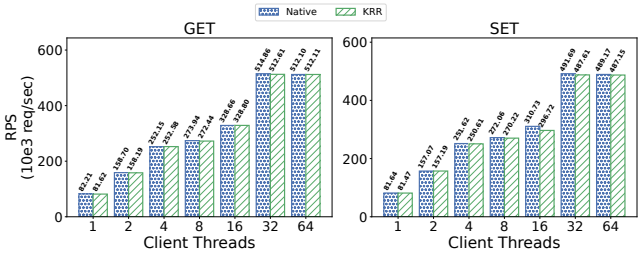
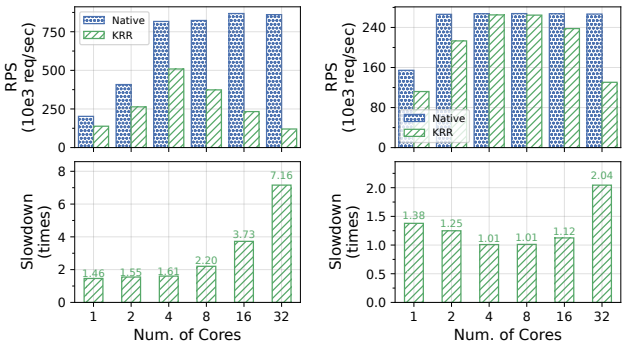
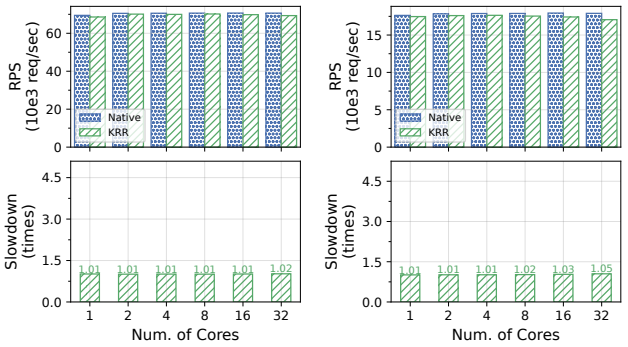


Figure 6: Redis Kernel-bypass Throughput



(a) File Size == 1KB (b) File Size == 4KB



(c) File Size == 16KB (d) File Size == 64KB

Figure 7: Throughput of Nginx kernel-bypass benchmarks with different file sizes.

the kernel-bypass setup, KRR does not need to record these transfer operations, resulting in minimal performance impact.

5.3 Reproduce kernel bugs using KRR

This section evaluates the effectiveness of KRR in recording and replaying Linux kernel bugs. To construct a representative evaluation dataset, we systematically collect bugs from two well-established bug-reporting sources. First, we gather bugs found by Syzbot [28], a Linux kernel continuous testing project that publicly reports kernel bugs along with reproducers (if any). Specifically, we identify 92 bugs that are found by Syzbot in Linux kernel 6.1.X and have associated reproducer programs. Next, we collect 81 bugs whose titles contain any of the keywords: *WARNING*, *BUG*, *deadlock*, and *KASAN* to focus on severe kernel errors. From these bugs, we randomly select 12 bugs. Next, to evaluate KRR on high-impact kernel vulnerabilities, we additionally collect 5 recent Linux kernel CVEs [4–8] that have high severity scores and were evaluated by prior research work [75] and rated at least as "Important Impact" by Red Hat Product Security [23]. The resulting dataset consists of 17 Linux kernel bugs.

Table 1 presents the bug reproduction results for 12 bugs found by Syzbot, along with the average iteration required to execute the reproducer before triggering the bug. First, 6 bugs (50%) are considered deterministic bugs because they can be reproduced consistently within the first iteration of the reproducer under native execution. KRR successfully reproduces all 6 deterministic bugs within a single iteration as well.

Second, the remaining 6 bugs are non-deterministic, requiring varying numbers of iterations to reproduce in native execution. Among these bugs, KRR is able to reproduce 5 of 6. Furthermore, our analysis of the median number of iterations required to first trigger the bug does not show a significant difference between the two approaches. In fact, in 3 out of the 6 non-deterministic cases, KRR reproduced the bugs with fewer iterations than the native execution.

For example, KRR performs better than native on bug #5. This bug is caused by a race condition, in which two threads

concurrently execute the function `gsm_cleanup_mux`, and accesses to the shared variable `dlci` need to occur in a specific order to trigger the use-after-free. KRR can reproduce this bug because the function acquires a mutex using `mutex_lock` halfway through its execution, giving a chance for other tasks to be scheduled even in a serialized execution given our algorithm (§3.3).

Bug #8, a potential deadlock in `rcu_report_exp_cpu_mult()`, can only be reproduced in a multi-core VM with native execution. This bug involves a lock ordering violation: when multiple cores contend for the same spinlock, one core acquires it with interrupts disabled (making it `hardirq-safe`). During this execution, the system triggers a BPF trace that attempts to acquire a `hardirq-unsafe` lock. This sequence—acquiring a `hardirq-safe` lock followed by a `hardirq-unsafe` lock—violates lockdep's locking rules and can lead to deadlock. The bug requires parallel execution: multiple cores must concurrently attempt to acquire the same lock with the interrupt disabled. This scenario cannot occur on single-core systems, where only one execution context can run at a time. Thus, KRR cannot reproduce this bug because its serialization approach prevents concurrent lock contention between cores.

Security-sensitive bugs. Table 2 shows the bug reproduction results for 5 Linux kernel CVEs. These vulnerabilities encompass a diverse range of bug types, including three privilege escalation vulnerabilities, a heap overflow vulnerability, and a control flow hijacking vulnerability. KRR can successfully record and replay all 5 CVEs, demonstrating its effectiveness in reproducing complex kernel exploitations.

ID	Description	Kernel	Det.	Repro.	Iterations Median	
					Native	KRR
#1	Paging error in vsyscall	6.1.77	N	Y	39.0	52.5
#2	null-ptr-deref in hugetlb	6.1.61	N	Y	22.0	21.0
#3	use-after-free in af_unix	6.1.62	N	Y	45.5	52.0
#4	null-ptr-deref in fs/sysv	6.1.34	Y	Y	1.0	1.0
#5	use-after-free in tty	6.1.35	N	Y	28.0	1.5
#6	Deadlock in ext3	6.1.31	Y	Y	1.0	1.0
#7	Deadlock in vfs_write	6.1.34	Y	Y	1.0	1.0
#8	Deadlock in bpf	6.1.84	N	N	115.0	N/A
#9	Deadlock in vhci	6.1.66	Y	Y	1.0	1.0
#10	Deadlock in af_unix	6.1.84	N	Y	209.0	187.0
#11	Warning in BPF	6.1.18	Y	Y	1.0	1.0
#12	Warning in BPF	6.1.86	Y	Y	1.0	1.0

Table 1: Reproduction results of bugs found by Syzbot. Some bugs are deterministic ("Det.") and all bugs except for one are reproducible by KRR ("Repro."). The table shows the median number of test iterations required to hit the bug across 10 runs.

CVE	Type	Kernel	Repro.
CVE-2024-1086	Privilege Escalation	6.1.0	Y
CVE-2022-0847	Privilege Escalation	5.10.222	Y
CVE-2022-0185	Heap Overflow	5.10.222	Y
CVE-2021-4154	Control Flow Hijacking	5.10.222	Y
CVE-2022-2639	Privilege Escalation	5.17.4	Y

Table 2: Summary of evaluated CVEs.

5.4 Storage Cost

KRR's storage cost is lower than VM-RR when recording kernel-bypass workloads because KRR does not need to record the data between the guest user-space and the hardware while VM-RR does. Across all evaluated workloads (§5.2.1), VM-RR on average records at a rate of 9.4 MB of data per second while KRR records 4.8 MB per second—the storage cost of KRR is 48.9% lower. The storage cost deduction also aligns with the low recording overhead of KRR on kernel-bypass workloads (§5.2).

When recording workloads that do not use kernel-bypass techniques, KRR imposes higher storage cost compared to VM-RR because it records additionally the software inputs. Specifically, for the RocksDB workloads (§5.1.1), VM-RR records an average of 8.26 MB of data per second, whereas KRR records 53.39 MB per second (546.57% more). In total, KRR and VM-RR on average record 999.22 MB and 464.48 MB of data, respectively. However, the recording data of KRR is highly compressible. After applying gzip compression, the recorded data of KRR is reduced to 144.52 MB (a 6.91x reduction), suggesting the potential for using hardware compression features [14] to compress the data in real-time—an optimization we leave for future work. These numbers are well within the bandwidth of data center storage, which reaches a few GB/s, so it can be easily sustained. Furthermore, the length of the trace can be reduced by taking frequent snapshots even on very long-running workloads.

5.5 Replay Performance

While KRR skips the userspace execution during the replay, which contributes significantly to reducing replay overhead, its replay execution is still generally slower than native execution by $20\times \sim 150\times$. This slowdown is expected given that our prototype's replayer uses the QEMU emulator (§3.5) in the single-step mode (one instruction per translation block). Based on our tests on the benchmarks in §5.1.1, with whole system execution, full emulation (TCG) alone introduces a $7.5\times \sim 20.6\times$ slowdown compared to native KVM. With single-step enabled, the slowdown reaches $120\times \sim 400\times$. KRR also disables QEMU TCG's optimization of lazy EFLAGS evaluation to simplify the logic that ensures consistency between the record and replay runs. Together, these factors contribute to the bulk of the replay over-

head in our prototype, but are not fundamental to the KRR's sliced record replay approach (§2).

Despite this performance trade-off, emulation-based replay is commonly used in existing record-replay systems [21, 40] for its rich debugging capabilities and portability. The periodic snapshots taken during recording also mitigate performance concerns by allowing replay to start from intermediate points rather than always replaying from the beginning. Future work could explore optimizations like enabling batch translation blocks [22] while maintaining replay accuracy to improve performance further under full emulation.

6 Discussion

Scalability. While the evaluation (§5.1.2, §5.1.1) shows that the scalability of KRR decreases with higher numbers of vCPU cores (e.g., beyond 8 cores), it demonstrates that KRR enables modern workloads to scale effectively on VMs with 1-8 vCPU cores—a sweet spot for most real-world deployments. This range aligns with industry standards, as exemplified by Azure's session host sizing guidelines, which designate 8 vCPUs as sufficient for heavy workloads [25].

Nevertheless, future work could enhance parallelism in KRR, moving beyond the current serialization of kernel execution. A promising direction is to integrate techniques from chunk-based serialization algorithms, such as Samsara's algorithm [54, 60]. Adopting such an approach in KRR could enable true parallel kernel execution and potentially broaden the class of bugs that KRR can reproduce.

Pass-through Device. KRR currently does not support pass-through devices or SR-IOV [26] that operate outside of kernel-bypass mode. This limitation is because KRR relies on hypervisor emulation to record device DMA buffers and timing. A potential solution involves leveraging KRR's in-guest recorder component. Since this recorder already operates within the guest kernel with privileged access to kernel data structures, it could be extended to intercept and log interactions with these pass-through devices. Following an approach similar to prior work [70], the in-guest recorder could monitor device driver operations, capturing DMA buffer contents and I/O timing information directly at the driver interface level rather than requiring hypervisor intervention.

Recording Scope. Without specialized hardware, no technique can record all non-deterministic failures – subtle timing changes can make such failures more likely or less likely to occur. KRR employs serialization to ensure deterministic execution of the kernel, which is crucial for accurate replay but can potentially mask certain concurrency bugs that depend on parallel thread interactions. Despite serializing the kernel execution, KRR's mechanism effectively models a single-core system where concurrency still occurs through context switching and interrupts. This is because KRR's serialization mechanisms allow for a change in which threads can grab the RC spinlock and run during preemption or interrupts, mirroring how the kernel's built-in schedule would change the running thread in response to such events. Consequently, KRR is still effective at reproducing a broad class of concurrency bugs that only manifest under specific thread interleavings. In essence, if a concurrency bug can be triggered on a single-core system through context switching, KRR can record and replay it.

On the other hand, by design and like other RR systems, KRR cannot reproduce concurrency bugs that strictly require true parallelism—i.e., the simultaneous execution of kernel code on multiple physical cores. This category includes bugs related to weak memory models or specific scenarios involving the parallel execution of uninterruptible kernel code on one core and normal kernel code on another. Furthermore, serialization may also make bugs that surface only within extremely narrow race windows, highly sensitive to precise parallel timing, more challenging to capture [50]. Despite the limitations in reproducing certain concurrency bugs, alternative approaches can be used to detect them. For instance, some concurrency bug detection techniques [52], such as the lockset data race detector [62], have a high degree of schedule independence, enabling the detection of bugs even in executions that do not manifest failures.

Privacy. Since KRR records user-space data to ensure correct kernel replay, this could raise potential privacy concerns. Like traditional settings, which assume hypervisors and kernels are trusted, we assume that KRR is trusted and privacy is not a design concern. This approach aligns with traditional VM RR approaches [60, 64], which also collect all VM data. However, our design has the advantage that it limits exposure by only recording data that directly interacts with the kernel (through system calls and memory accesses). Future work could explore selective data filtering [34] or trace encryption for use cases where privacy guarantees are important.

7 Related Work

Reproducing the execution in which a failure occurs is essential for developers to diagnose bugs in system software. Two main approaches have emerged to address this challenge: record and replay techniques that capture and reproduce the exact execution, and partial reconstruction methods that recreate critical aspects of the execution without full recording.

Application RR. Application record-replay techniques record non-deterministic inputs to the application and its internal non-determinism such as concurrency. Among them, Mozilla RR [56] provides deterministic record-replay for multi-threaded applications. Similar to KRR, RR records the concurrency non-determinism by serializing concurrent threads and recording the schedule. Castor [53] focuses on improving the recording performance on multi-threaded but data-race-free applications. Specifically, Castor allows user-space threads to run

in parallel but records the ordering of the thread synchronizations. During the replay, Castor reproduces the execution by replicating the synchronization ordering. However, this approach is impractical for recording the kernel due to the prevalence of data races in operating systems [67].

Whole-machine RR. Whole-machine record-replay techniques record and replay the entire VM execution by capturing all non-deterministic events across the VM boundary. Retrace [64] introduces the first whole-machine RR tool based on the hypervisor, which is adopted later by VMWare in its first Fault Tolerance implementation [63]. More recently, PANDA [40] develops a whole-machine RR platform with powerful extensions for binary analysis. However, they all focus on single-core systems. Multi-core record and replay has evolved through several key systems. SMP-Revirt [42] introduced CREW (concurrent-read, exclusive-write) protocols for recording memory access ordering. ReEmu [35] enhanced CREW's scalability based on full system emulation. Samsara [60] improved efficiency by leveraging hardware assistance and employing chunk-based memory access recording.

Failure Reconstruction. Prior work [36, 38, 39, 57, 66, 71, 76] explores failure diagnosis without record-replay. REPT [38] leverages Intel PT [12] to record both program control flow and timing information, along with the final memory dump. It then tries to reconstruct the execution history through binary analysis of these recorded traces. However, it faces two important limitations: it is not fully accurate and cannot reconstruct all data values that are necessary for debugging, struggling, in particular, to reconstruct long executions. Kernel REPT [45] extends this to kernel failures with more extensive analysis functionality. In contrast, KRR provides a more efficient and scalable solution for kernel-level record-replay, enabling reliable reproduction of failures.

8 Conclusion

This paper presents KRR, an efficient and scalable kernel record-replay system. Unlike the traditional whole-VM RR approach, KRR incurs low overhead on the VM performance because it reduces the RR boundary from the entire VM to the guest kernel only, completely avoiding the expensive yet unnecessary cost that whole-VM RR has to pay for recording the guest user space. The evaluation shows that KRR achieves high recording performance on several real-world workloads, and scales well to multi-core VMs. On modern workloads that exploit kernel-bypass, KRR even shows close-to-native recording performance. We believe KRR fills the gap between application RR and whole-VM RR, and we hope KRR can help developers efficiently and effectively diagnose failures in the kernel.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their feedback, which greatly improved the paper. We also thank the Purdue Reliable and Secure System Lab members for their helpful comments on this work and earlier drafts. We are also grateful to Cloudlab for providing experimental machines. This work was funded in part by National Science Foundation (NSF) under grants CNS-2140305 and CNS-2145888 and gifts from Google and Intel.

References

[1] Background Snapshot. https://kvm-forum.qemu.org/2021/kvm2021-background-snapshot.pdf. Last accessed: April, 2025.

[2] Benchmarking tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Last accessed: April, 2024.

[3] CloudLab. https://www.cloudlab.us/. Last accessed: April, 2025.

[4] CVE-2021-4154. https://access.redhat.com/security/cve/cve-2021-4154. Last accessed: November, 2024.

[5] CVE-2022-0185. https://access.redhat.com/security/cve/cve-2022-0185. Last accessed: November, 2024.

[6] CVE-2022-0847. https://access.redhat.com/security/cve/cve-2022-0847. Last accessed: November, 2024.

[7] CVE-2022-2639. https://access.redhat.com/security/cve/cve-2022-2639. Last accessed: November, 2024.

[8] CVE-2024-1086. https://access.redhat.com/security/cve/cve-2024-1086. Last accessed: November, 2024.

[9] DPDK. https://www.dpdk.org/. Last accessed: April, 2024.

[10] f-stack. https://github.com/F-Stack/f-stack. Last accessed: November, 2024.

[11] GDB Reverse Debugging. https://sourceware.org/gdb/wiki/ReverseDebug. Last accessed: April, 2025.

[12] Intel PT. https://edc.intel.com/content/intel-processor-trace/. Last accessed: November, 2024.

[13] Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Last accessed: April, 2025.

[14] Intel® QuickAssist Technology (QAT) QATzip Library. https://intel.github.io/quickassist/qatlib/qatzip.html. Last accessed: November, 2024.

[15] io_uring. https://man7.org/linux/man-pages/man7/io_uring.7.html. Last accessed: November, 2024.

[16] Kernel GDB Debugging. https://www.kernel.org/doc/html/v4.16/dev-tools/gdb-kernel-debugging.html. Last accessed: November, 2024.

[17] Kernel Parameters. https://docs.kernel.org/admin-guide/kernel-parameters.html. Last accessed: April, 2024.

[18] Kernel rseq. https://github.com/torvalds/linux/blob/master/kernel/rseq.c. Last accessed: November, 2024.

[19] NAPI. https://docs.kernel.org/networking/napi.html. Last accessed: November, 2024.

[20] QEMU. https://www.qemu.org/. Last accessed: April, 2024.

[21] QEMU Record Replay. https://www.qemu.org/docs/master/system/replay.html. Last accessed: April, 2024.

[22] QEMU Translation. https://www.qemu.org/docs/master/devel/tcg-ops.html. Last accessed: April, 2025.

[23] Red Hat Security Ratings. https://access.redhat.com/security/updates/classification/. Last accessed: November, 2024.

[24] Redis Benchmark. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/. Last accessed: November, 2024.

[25] Session host virtual machine sizing guidelines. https://learn.microsoft.com/en-us/windows-server/remote/remote-desktop-services/virtual-machine-recs. Last accessed: November, 2024.

[26] SR-IOV. https://www.ibm.com/docs/en/power10?topic=networking-single-root-io-virtualization. Last accessed: April, 2025.

[27] Storage performance development kit application event framework. https://www.intel.com/content/www/us/en/developer/articles/technical/storage-performance-development-kit-application-event-framework.html. Last accessed: December, 2024.

[28] Syzbot. https://syzkaller.appspot.com/. Last accessed: November, 2024.

[29] Time-of-check to time-of-use. https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use. Last accessed: November, 2024.

[30] Unreliable guide to hacking the linux kernel. https://www.kernel.org/doc/html/v4.13/kernel-hacking/hacking.html. Last accessed: April, 2025.

[31] wrk. https://github.com/wg/wrk. Last accessed: April, 2024.

[32] Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. Veil: A protected services framework for confidential virtual machines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 378–393, Vancouver, Canada, March 2023.

[33] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[34] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS08, page 319–328. ACM, March 2008.

[35] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, page 207–218. ACM, February 2013.

[36] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009.

[37] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, March 2011.

[38] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse debugging of failures in deployed software. In

13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 17–32, Carlsbad, CA, October 2018. USENIX Association.

[39] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. RETracer. In *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, May 2016. ACM.

[40] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[41] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. Disaggregating rocksdb: A production experience. *Proceedings of the ACM on Management of Data*, 1:1 – 24, 2023.

[42] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, page 121–130, New York, NY, USA, 2008. Association for Computing Machinery.

[43] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6, 2012.

[44] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, Broomfield, CO, October 2014. USENIX Association.

[45] Xinyang Ge, Ben Niu, and Weidong Cui. Reverse debugging of kernel failures in deployed systems. In *USENIX Annual Technical Conference*, 2020.

[46] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 66–83, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 35–51, New York, NY, USA, 2023. Association for Computing Machinery.

[48] Sishuai Gong, Wang Rui, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowplow: Effective kernel fuzzing with a learned white-box test mutator. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1124–1138, New York, NY, USA, 2025. Association for Computing Machinery.

[49] Shreyas Kharbanda and Pedro Fonseca. Always-on recording framework for serverless computations: Opportunities and challenges. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies (SESAME'23)*, pages 41–49, Rome, Italy, 2023.

[50] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380. USENIX Association, August 2021.

[51] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2078–2095, 2022.

[52] Congyu Liu, Sishuai Gong, and Pedro Fonseca. Kit: Testing os-level virtualization for functional interference bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '23, page 427–441. ACM, January 2023.

[53] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 693–708, New York, NY, USA, 2017. Association for Computing Machinery.

[54] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. *ACM SIGARCH Computer Architecture News*, 36(3):289–300, June 2008.

[55] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. An in-depth analysis of duplicated linux kernel bug reports. In *Proceedings 2022 Network and Distributed System Security Symposium*, NDSS 2022. Internet Society, 2022.

[56] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017.

[57] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. PRES. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, New York, NY, USA, October 2009. ACM.

[58] Dinglan Peng, Congyu Liu, Tapti Palit, Anjo Vahldiek-Oberwagner, Mona Vij, and Pedro Fonseca. Pegasus: Transparent and unified kernel-bypass networking for fast local and remote communication. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 360–378. ACM, March 2025.

[59] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), nov 2015.

[60] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. Samsara: efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 551–564, USA, 2016. USENIX Association.

[61] Robin Salkeld, Wenhao Xu, Brendan Cully, Geoffrey Lefebvre, Andrew Warfield, and Gregor Kiczales. Retroactive aspects: programming in the past. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, ISSTA '11, page 29–34. ACM, July 2011.

[62] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[63] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, December 2010.

[64] MXVMJ Sheldon and Ganesh Venkitachalam Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007.

[65] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo workarounds for kernel bugs. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2381–2398. USENIX Association, August 2021.

[66] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with Hardware-Enhanced Post-Crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 17–32, Vancouver, BC, August 2017. USENIX Association.

[67] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.

[68] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture - ISCA '03*, ISCA '03. ACM Press, 2003.

[69] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2313–2328. ACM, October 2017.

[70] Hyunmin Yoon, Shakaiba Majeed, and Minsoo Ryu. Exploring os-based full-system deterministic replay. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC 2018, page 1077–1086. ACM, April 2018.

[71] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10. ACM, April 2010.

[72] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 131–146, New York, NY, USA, 2019. Association for Computing Machinery.

[74] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 540–555. ACM, April 2021.

[75] Xiaochen Zou, Yu Hao, Zheng Zhang, Juefei Pu, Weiteng Chen, and Zhiyun Qian. Syzbridge: Bridging the gap in exploitability assessment of linux kernel bugs in the linux ecosystem. In *Proceedings 2024 Network and Distributed System Security Symposium*, NDSS 2024. Internet Society, 2024.

[76] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution reconstruction: harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21. ACM, June 2021.

Deterministic Client: Enforcing Determinism on Untrusted Machine Code

Zachary Yedidia
Stanford University

Geoffrey Ramseyer
Stanford University and
Stellar Development Foundation

David Mazières
Stanford University

Abstract

This paper presents Deterministic Client (DeCl), a software-based sandboxing system for enforcing deterministic behavior on untrusted machine code, either x86-64 or Arm64. DeCl adapts techniques from Software Fault Isolation (SFI) traditionally used to guarantee memory isolation to instead enforce the stronger property of determinism. By using a simple and efficient machine code verifier that can guarantee that a program behaves deterministically, DeCl does not rely on a trusted compiler/interpreter for correctness. This allows the use of LLVM without compromising the size of the trusted code base. We also describe how to implement two efficient metering mechanisms that enforce deterministic preemption of sandboxed programs, and how DeCl can be implemented in combination with traditional software-based isolation, by making the sandboxed code position-oblivious. DeCl is able to combine and improve upon the benefits of both interpreters and JIT compilers at once, with low CPU overhead, fast startup time, and strong security via a small trusted code base. We evaluate DeCl's effectiveness on general-purpose CPU benchmarks, as well as in an application-specific context by integrating with the Groundhog smart contract engine, and using DeCl for zero-knowledge-proof verification.

1 Introduction

Program determinism is essential for executing smart contracts correctly. Smart contracts are programs that are uploaded by completely untrusted parties, and yet must be guaranteed to perform the same set of operations every time they are run by an honest node. Since smart contract programs increasingly form the backbone of digital currency systems, it is important for them to be performant. Existing approaches to smart contract sandboxing are language-based: they define an intermediate language (IL) with deterministic semantics, such as WebAssembly or EVM bytecode, and then execute the language via a trusted interpreter or just-in-time (JIT) compiler. Unlike with sandboxes that enforce memory isolation, this language-based approach is the only existing method for enforcing determinism in an adversarial setting. Approaches for enforcing determinism must necessarily be software-based, as commodity hardware does not provide sufficient mechanisms for hardware-based enforcement.[1]

Enforcement	Memory isolation	Determinism
Language	WebAssembly, eBPF, JavaScript, ...	WebAssembly, eBPF, EVM Bytecode, ...
Binary analysis	LFI, NaCl, PittSFIeld, ...	**DeCl**
Hardware	Pagetables, Segmentation	–

Table 1: Approaches that enforce memory isolation and determinism via hardware-based or software-based methods.

This paper introduces Deterministic Client (DeCl), a new software-based approach to enforcing determinism. By adapting binary analysis techniques from Software Fault Isolation (SFI) [36, 50, 54, 57], we design a machine code analyzer that can determine if a program is part of a deterministic subset of either x86-64 or Arm64. If so, the program is guaranteed to be deterministic, and can be directly run as native code, without any intermediate translation step. This design significantly reduces runtime CPU overhead and startup latency, and removes the need for a trusted translator. Highly optimizing ahead-of-time compilers such as LLVM can be used without needing to trust them. In order to generate programs that can pass verification, we develop a program rewriter that operates on assembly files, allowing us to rewrite the output of LLVM/GCC and therefore support many existing languages and compiler optimizations.

Binary analysis techniques from SFI are well-known for enforcing memory isolation, but this is the first work to apply

[1] Both x86-64 and Arm64 have cases of instructions that cause unpredictable behavior that is not trapped by hardware. Existing hardware cannot prevent these instructions from executing.

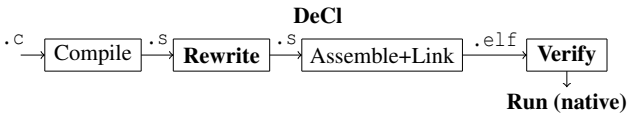


Figure 1: The DeCl compilation flow. Determinism properties are enforced directly on the machine code output of a compiler. In order to produce binaries that pass verification, a rewriter is used during compilation. Non-bold steps use existing compiler infrastructure.

them to determinism. Table 1 summarizes existing sandboxing approaches for both memory isolation and determinism, and Figure 1 shows the overall architecture of DeCl, which is typical of SFI systems. We describe three main components:

1. Deterministic instruction enforcement: programs that attempt to use non-deterministic instructions must be rejected. We describe cases of non-determinism in x86-64 and Arm64, along with how to guarantee their absence.

2. Deterministic metering: programs cannot be permitted to run indefinitely. However, a traditional timer interrupt on its own may cause the program to terminate after different side effects each time the program runs. We describe two mechanisms for deterministic preemption.

3. Integration with software-based memory isolation: DeCl can be used in conjunction with any memory isolation mechanism (hardware-based or software-based). We describe additional changes that are necessary in order to use DeCl with LFI [57], a software-based sandbox where all sandboxes share an address space. This makes it possible to use DeCl sandboxes within a Linux process and enables fast sandbox startup times.

We evaluate DeCl's performance in terms of CPU overhead and startup time. We measure both general benchmarks (SPEC 2017) and benchmarks specific to smart contracts. With both metering and software-based isolation, DeCl only imposes a modest $\sim 20\%$ overhead with metering, measured on a subset of SPEC 2017 for both x86-64 and Arm64.

We integrate DeCl into Groundhog [45], a scalable smart contract engine currently using a WebAssembly interpreter. Using native code for smart contracts allows them to run CPU-intensive computations faster, by roughly $2\times$ (compared to JIT compilation) or $30\times$ (compared to interpretation). Since some important smart contracts are short-lived, we have optimized the system to have a sandbox startup latency of less than $15\mu s$ (load and execute), and we ensure that loading sandboxes does not interfere with Groundhog's scalability.

The rest of the paper describes the design of DeCl for both x86-64 and Arm64. We find that properties of x86-64 such as undefined flags, variable-length instructions, a small register file, flags-based conditional branches, and stack-based `call`/`ret` are most responsible for higher complexity compared to Arm64.

2 Background

2.1 Threat Model

DeCl enforces determinism against adversarially-constructed programs. An untrusted party provides a program (expressed as x86-64 or Arm64 machine code), and the DeCl verifier accepts it only if it can be guaranteed to be deterministic. The verifier is allowed to reject programs if it cannot confirm determinism, but nonetheless, we would like to be able to accept as wide a variety of programs as possible, while imposing minimal performance overhead. The determinism guarantee only applies within a particular architecture, but covers all valid microarchitectures. For example, a program validated for DeCl-Arm64 will behave deterministically on all Armv8.0-compliant processors, including virtual processors such as QEMU. In certain cases, we require a processor to have certain architectural extensions, such as the x86-64 BMI2 extension for the `SHRX` instruction. Existence of extensions like these can be probed using `cpuid` (or equivalent). Requiring extensions reduces the number of processors that can be used to run the program (older processors become excluded), but allows for newer instructions to be used.[2] We rely on the hardware being correct for the ISA-subset accepted by the verifier.[3]

We also support requiring that the programs that run inside DeCl are *metered*, and therefore cannot use more than a limited amount of CPU time. This requirement, when combined with the determinism requirement, requires special design, since the preemption must happen in a deterministic way. The runtime system is initialized with a limit on the number of instructions that can be executed before preemption occurs.

We assume the following properties about the runtime system, which are typical for adversarially-deterministic systems such as smart contract engines:

1. The runtime provides a deterministic "runtime call API" that a sandboxed program uses to have external effects.

2. The runtime disables shared-memory multiprocessing.

3. The runtime deterministically initializes the sandbox by zeroing the initial memory. Section 7.2.1 discusses the performance impact on startup time.

Enforcing determinism requires that the sandboxed program must not be able to access memory outside its sandbox (memory isolation). DeCl can work with either hardware or software-based memory isolation, and we describe additional

[2] Older processors could be supported via dynamic recompilation, but we do not implement or evaluate this approach.

[3] Recently discovered hardware issues we are aware of [1, 3, 5] involve programs that are already disallowed by the verifier by default.

changes needed for the software-based case in Section 5. We also terminate programs after a memory fault. This is relevant due to *torn stores*: stores that straddle a mapped and unmapped page boundary. On Arm64, it is implementation-defined whether the memory in the mapped region is altered or not. While there are multiple mechanisms to solve this issue, such as simulating the stores, or performing a load before every store [20], we simply terminate the program.

Side channels Side-channel attacks exploit non-determinism in the runtime API or instruction behavior to recover information about parts of the system that should not be available to the application. These exploits most commonly use timers to recover information about data in the cache. Since DeCl removes all non-determinism in the runtime API and available instruction set, timers or other side-channel vectors (e.g., parallelism or atomics [59]) are impossible to use.

2.2 Secure Machine Code Analysis

The verifier must be able to guarantee that the program cannot execute any instructions that have not been analyzed. Binary analysis systems that enforce memory isolation, such as NaCl [50] and LFI [57], contend with the same issue. Hence, we can use the same techniques they use, which we summarize below.

On Arm64, thanks to fixed-width instructions, it suffices to enforce that memory is never executable and writable at the same time (commonly called W^X). As a program cannot write to executable regions and it cannot jump to the middle of an instruction (which causes a trap), all instructions can be verified before making memory executable.

On x86-64, in addition to W^X memory, we must prevent untrusted code from jumping to the middle of an instruction and executing a completely different instruction sequence from the one disassembled. The solution we use is *aligned bundles*, first introduced by the PittSFIeld [36] system, and now supported by both the LLVM and GCC assemblers. Instructions are organized into 32-byte bundles, as illustrated in Figure 2. The verifier rejects any program that includes instructions that span bundle boundaries, so a compiler targeting DeCl adds `nop` padding to move these into the next bundle. Additionally, the verifier ensures that all jumps target bundle boundaries—static branches are directly inspected, and indirect branches (including returns, which are rewritten into indirect branches) are required to include a preceding mask instruction that aligns the target to a bundle boundary by zeroing the bottom 5 bits of the target.

In both cases, programs must separate code and data, lest the data correspond to unsafe instructions that would be rejected by the verifier. Fortunately, modern x86-64 and Arm64 compiler toolchains already can generate separate code and read-only data segments by default. The `-z separate-code` linker flag can be used to instruct the linker to create distinct code and data segments in the final binary.

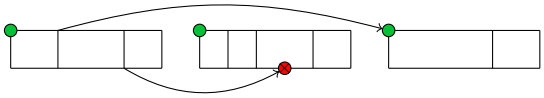


Figure 2: Illustration of aligned bundles. Each bundle consists of multiple instructions, padded such that no instruction spans a bundle boundary. Jump targets must be bundle-aligned (indicated by green dots). The verifier rejects jumps that may target the middle of a bundle or instruction.

With these methods for guaranteeing secure machine code analysis in hand, we can design machine code analyzers that only accept programs with known instructions whose semantics are fully understood by the analyzer. Programs containing any unknown instructions are rejected. Traditionally, this technique has been used to make verifiers that only accept programs with instructions known to be memory-safe, but we show that this can also be applied to accept only programs with instructions known to be deterministic.

3 Deterministic Machine Code

The DeCl verifier must only accept programs that execute deterministically. Therefore, these programs must contain only instructions known to be deterministic. Additionally, some x86-64 instructions execute with unpredictable semantics only for certain inputs. To accept these (useful) instructions, the verifier must ensure that these problematic inputs are not present when those instructions execute. Enforcement on Arm64 is simpler because the ISA has fewer cases of unpredictable behavior. We describe our implementation for Arm64 and x86-64 separately.

Floating point We restrict our analysis to the integer subsets of x86-64 and Arm64, including SIMD but not including floating point. In particular, instructions like `rsqrtss` (Reciprocal of Square Root) are required to provide at most some amount of error but within those bounds can use any precision, which can cause non-determinism. We believe a deterministic subset of floating point operations could be safely included in DeCl (including things like FP addition, but excluding `rsqrtss`), but leave this out of scope in this paper.

3.1 Arm64

In Arm64, the following cases of non-determinism exist at the instruction level:

- Instructions encoded with malformed SBZ/SBO (should-be-zero/should-be-one) fields are UNPREDICTABLE.

- Some instructions have explicitly non-deterministic semantics, such as instruction pairs for atomic loads and stores. For example, the instruction `stxr` returns different values depending on the status of the exclusive monitor, which can depend on factors such as timer interrupts generated for the host kernel. As another example, memory instructions that both increment and write back to the same register are UNPREDICTABLE: `ldr x0, [x0], #16`. Early versions of LLVM could erroneously produce this instruction [17].
- Unallocated instructions.

Our verifier rejects programs that contain such instructions, as well as instructions that are UNDEFINED,[4] with one exception: we allow the `udf #0` instruction as an explicit way to generate an undefined instruction exception.

The DeCl Arm64 subset only supports instructions from the Armv8.0 base ISA, with roughly 180 base instructions and an additional 430 SIMD instructions. These instruction counts are derived from the decoder we use, so different counts could be expected if using a decoder that classifies distinct instructions differently.

Arm64 also allows hardware-enforced stack pointer alignment. When enabled by the OS, stores to the stack pointer when it is not aligned to a 16-byte boundary cause a fault. While we could simply require that this check always be enabled (as is done by Linux), our verifier also requires that modifications to `sp` preserve its alignment. Arm64 also supports multiple base page sizes (4KiB, 16KiB, or 64KiB), and for determinism DeCl must pick one. A machine with a smaller page size can efficiently emulate one with a larger page size, so picking the largest page size is best. Our implementation uses 16KiB pages because there isn't any commodity hardware configured with 64KiB pages.

3.2 x86-64

The situation on x86-64 is more complex, as the instruction encoding space is far larger. We start by limiting the instructions accepted by the verifier to a known, fully-enumerable, subset of x86-64. We recorded all instructions that can be encoded by the Fadec encoder [18] and represent this set with a binary decision diagram (BDD), which can be stored in less than 200KiB. Instructions accepted by the verifier must first be decoded by the BDD, guaranteeing that the verifier operates only on instructions with known semantics. Once approved by the BDD, the instructions are further restricted to the DeCl subset, which includes roughly 100 instructions from the base x86-64 ISA, along with 125 SIMD instructions from the SSE2 extension. For branch-based metering, we also require the `SHRX` instruction from the BMI2 extension.

[4] Even though according to the Arm manual, UNDEFINED instructions deterministically generate an "undefined instruction" exception.

Instruction	Undefined if	Ex. guarded sequence
`SHLD`, `SHRD`	`COUNT > OPSIZE`	`andb $0x1f, %cl` `shrdl %cl, %ebx, %eax`

`BSR`, `BSF`	`SRC = 0`	`test %eax, %eax`
		`jz .SKIP`
		`bsr %eax, %ebx`
		`.SKIP:`

Table 2: x86-64 instructions that may produce undefined results for certain inputs, and guards that can be applied beforehand to rule out this possibility. The guard sequence for shifts may additionally require saving and restoring `%rcx` (because `%cl` is modified), which can be done using the rewriter-reserved `%r11` register.

Moreover, some instructions may generate *undefined* (i.e., non-deterministic) values. For example, the `SHLD` instruction produces an undefined value if the shift operand is larger than the bit-width of the shifted value. Many instructions also put flags into undefined states. For example, the `BT` (bit-test) instruction modifies the CF flag, leaves the ZF flag unchanged, and makes the OF, SF, AF, and PF flags undefined. Observing the state of an undefined flag can cause non-deterministic behavior. While individual CPU implementations are typically deterministic in this respect, the behavior is not consistent across CPU microarchitectures. A program that abuses an undefined flag will behave differently when run on Intel vs. AMD processors, and in some cases even when run on P-cores vs. E-cores within the same processor, since P- and E-cores do not necessarily share the same microarchitecture.

3.2.1 Preventing undefined results

The instructions in Table 2 may produce undefined results [26, 27]. Our program rewriter adds one or more guard instructions to force the input to be valid, and the verifier checks for the presence of these guards before marking memory executable. For example, for a shift of 16-bit data, we apply a mask to the shift amount to force it to be less than or equal to 16.

As of October 2024, the Intel SDM has been updated to say that if the content source of a `BSR`/`BSF` is zero, the result is unmodified, rather than undefined. However, the pseudocode of the operation continues to say the result is undefined, and other parts of the manual (documentation for `TZCNT`/`LZCNT`) still refer to the result as undefined in the case of a zero input. We continue to consider the output as undefined for `BSR`/`BSF` instructions with an input of zero.

3.2.2 Preventing use of undefined flags

Blocking all instructions that may read or write undefined flags would unreasonably restrict the set of verifiable programs. Instead, our verifier reasons about the data-flow of the

Algorithm 1 Flags Analysis

```
1:  function ANALYZE(B, Report)
2:      U ← IN[B]                          ▷ Set of undefined flags
3:      for all instruction I in basic block B do
4:          if Report and I reads a flag f and f ∈ U then
5:              Raise error
6:          end if
7:          for all flags f defined by I do
8:              U[f] ← 0
9:          end for
10:         for all flags f undefined by I do
11:             U[f] ← 1
12:         end for
13:     end for
14:     OUT[B] ← U
15: end function
16: function ANALYZECFG(CFG)
17:     for all basic block B in CFG do
18:         IN[B] ← 1*
19:     end for
20:     for all basic block B in CFG do
21:         ANALYZE(B, false)
22:     end for
23:     for all basic block B in CFG do
24:         IN[B] ← OUT[B_p1] | ... | OUT[B_pn]
25:     end for
26:     for all basic block B in CFG do
27:         ANALYZE(B, true)
28:     end for
29: end function
```

flags, so that it can verify whether any instruction that reads a flag does not observe an undefined value.

Although indirect branch targets are not statically known, the verifier enforces the target bundle-alignment requirement via a masking `and` instruction directly before an indirect jump. This instruction leaves every observable flag in a defined state[5], so we need only reason about data-flow of flags on the part of the control-flow graph of the program that is defined by static branches.

We define a function called ANALYZE (Algorithm 1) that runs on each basic block and uses a set IN, representing the set of flags that are undefined at entry to the basic block, to compute OUT, the set of flags that are undefined after executing the basic block. ANALYZE steps through each instruction in the basic block, updating its set of undefined flags initialized from IN, and signals an error on instructions that appear to read an undefined flag. Once complete, it updates the OUT set for the corresponding basic block. In each *iteration*, ANALYZE is run on every basic block. In a full analysis, iterations are run until the program passes verification, or until a fixpoint. In practice, we have have found that all conforming programs we have compiled with LLVM or GCC pass verification within 2 iterations. Thus it is reasonable to stop analysis and reject the program after 2 iterations if the verifier can not yet prove that no undefined flags are read. ANALYZE uses an internal table adapted from the Intel manual [28] that specifies for each x86-64 instruction whether it reads, modifies, or undefines a flag.

This analysis may reject programs that do not read undefined flags, but it will never accept a program that reads undefined flags. A program can always be written such that it passes verification in only one iteration by always setting flags in the same basic block where they are read. The cross-basic block analysis is only necessary to support code generation behavior in existing compilers (as we want to avoid modifying LLVM/GCC).

3.2.3 Spurious prefixes

CPU implementations consistently accept instructions with unnecessary prefixes, which are ignored (but have caused security bugs [5]). Our verifier only accepts programs that do not use spurious prefixes—this is a side effect of the fact that the verifier only accepts instructions that can be produced from an x86-64 encoder. We make an exception for `nop` instructions, which are often encoded with additional prefixes for padding and have canonical representations.

3.3 CPU Fuzzing

In order to gain confidence in the system beyond just the information in architectural reference manuals and formal ISA semantics, we have a fuzzer that distributes programs to machines of various microarchitectures. The fuzzer is not a necessary component of the DeCl system, but is used as a testing mechanism to find errors in the hand-written verifier. Ultimately, the verifier is the trusted component that must determine if a program is guaranteed to be deterministic or not.

The fuzzer randomly samples instructions from the ISA (for Arm64 this is the set of 4-byte integers; for x86-64 this is the set represented by the BDD used to restrict instructions to a known encodable subset). These instructions are used to create small basic blocks until they pass verification and then concatenated into verified programs consisting of megabytes-to-gigabytes of code. These code blocks are then executed in an execution environment that records the final state of the registers and memory. The verifier can be configured to allow memory accesses that are masked to target a certain memory region (the fuzzer will map pages in the region as they are accessed). Our fuzzer currently does not generate any branch instructions, but this is not a fundamental limitation—it is simpler to disallow branches during fuzzing, and since there are very few branch instructions (on the order of 10), we can

[5] AF is undefined by `and`, but AF cannot be observed by any instruction in DeCl's x86-64 subset.

manually analyze their behavior.

We have run our fuzzer on several (5+) microarchitectures for both Arm64 and x86-64. In our testing, the fuzzer quickly finds cases of non-determinism caused by undefined flags or undefined results if we allow such cases to pass verification.

4 Deterministic Metering

By default, there are no constraints on the amount of CPU time that a DeCl program may consume. To prevent a program from taking over the CPU, we use a notion of "metering," (commonly called "gas"). The program continues running until there is no more gas in the meter, at which point it is preempted. A typical operating system would implement this by using timer interrupts. However, that design is not deterministic, since the interrupt may fire non-deterministically before or after a runtime call with side-effects. In the case of atomic transactions, aborting after a timeout could cause replicas to disagree about whether a transaction succeeded or ran out of gas.

A deterministic alternative is to keep an instruction count. By terminating after N instructions have executed, programs can be deterministically preempted. Unfortunately, the performance monitoring unit (PMU) in modern processors does not provide deterministic instructions counts. As a result, we must manually track the number of executed instructions (the amount of "gas" used) via additional instructions that must be included in a sandboxed program. At a high level, these extra instructions debit a gas counter register (`x23` or `%r12`) according to the number of instructions executed in a basic block and use this counter to cause deterministic termination, either via a trap/branch into the DeCl runtime or via preemption from a timer. We will refer to these two mechanisms as *branch-based* and *timer-based* metering respectively. While timer-based metering makes use of a non-deterministic timer, its use in combination with the gas counter allows for deterministic preemption (details in Section 4.3).

Furthermore, since we are enforcing metering directly on machine code programs, the metering schemes must be efficiently checkable by our static verifier. The verifier must be certain that there is no way to execute a branch instruction without also executing the immediately preceding metering instructions. This could happen if a previous branch or jump skipped over the metering instructions and directly targeted a branch. We use *aligned bundles* to prevent this scenario.[6]

4.1 Aligned Bundles

With metering enabled, we require aligned bundles (discussed earlier in Section 2.2) on both Arm64 and x86-64. To implement bundling on Arm64, we reserve the `x24` register for holding bundle-aligned addresses. The verifier will only accept indirect branches that target `x24`, and will only accept instructions that modify `x24` by zeroing the bottom $\log(N)$ bits from the source register, where N is the bundle size (e.g., `bic x24, xN, 0xf`). All branches must target the beginning of a bundle, making it impossible to execute instructions at the end of a bundle without executing those at the start.

On Arm64, we use 16-byte (4-instruction) bundles for branch-based metering, and 8-byte (2-instruction) bundles for timer-based metering. On x86-64, we use the 32-byte bundles that are already required for machine code verification.

4.2 Branch-based Metering

Branch-based metering keeps an instruction meter in a reserved register (`x23` or `%r12`). The verifier enforces that every basic block ends with instructions that decrease the meter by the number of instructions in the block, and checks that the meter has not reached zero.

In general, detecting basic blocks in machine code is not possible because indirect branches may jump anywhere in the program. However, we do not need precise basic block analysis in order to provide metering, so long as any imprecision is conservative and deterministic. Consider a program without indirect branches; in this case all basic blocks are known statically, and we can charge the correct amount of gas at the end of each basic block. Now, if the program includes indirect branches, those branches may arrive somewhere within an existing basic block. The program will then be charged as if the entire basic block executed, even if it was only partially executed due to an indirect branch arriving in the middle. This may deterministically overcharge the program, but that is safe and only means the program terminates sooner. A program can remove this imprecision by splitting its basic block at indirect branch targets. Splitting a basic block is always legal.

A metering epilogue, shown in Fig. 3, must be inserted at the end of every basic block. This epilogue decreases the instruction meter and then checks if the top bit of the count is 1. If so, the meter has underflowed and a trap instruction is executed. Due to the encoding of the subtract instruction, it is impossible to both underflow and have a zero at bit 63 of `x23`/`%r12` (the immediate cannot be large enough).

Preserving flags On Arm64 and x86-64, many branch instructions jump based on the status of the flags. A gas check inserted before a branch cannot, therefore, modify the flags. Arm64 provides the `tbz` instruction, which performs a conditional branch based on a bit test rather than a flag. On x86-64, the only conditional branch that does not read flags is `jrcxz`, which performs a jump if `%rcx` is zero. We use `jrcxz` to build a metering sequence that preserves flags (Fig. 3a). We must also use an `lea` instruction in order to subtract from the gas register without modifying any flags, and a flags-preserving shift (`shrx`, provided by the BMI2 extension). One possible

[6] It is also possible to use hardware-enforced control-flow (BTI on Arm64 and IBT on x86-64), but these mechanisms are not yet widely deployed.

```
lea -n(%r12),%r12
push %rcx
mov $0x3f,%ecx
shrx %rcx,%r12,%rcx
jrcxz OK
int3
OK: pop %rcx
<branch>
```

(a) x64 flags preserved

```
sub $n, %r12
jns OK
int3
OK: <branch>
```

(b) x64 flags clobbered

```
sub x23, x23, #n
tbz x23, #63, OK
brk #0
OK: <branch>
```

(c) Arm64

Figure 3: Instruction sequences for branch-based metering, where n denotes the number of instructions in the immediately preceding basic block. The x86-64 sequence is complicated for cases that must avoid modifying the flags register.

```
sub x23, x23, #n
<branch>
```

(a) Arm64

```
lea -n(%r12), %r12
<branch>
```

(b) x64

Figure 4: Instruction sequences for timer-based metering on Arm64 and x86-64.

optimization would be to detect the preceding instruction that sets the flags and, if possible, pull that instruction into the metering sequence directly before the branch—avoiding the need to preserve flags. We do not perform this optimization.

4.2.1 Removing gas checks

The expensive part of the basic block epilogue is the underflow check. We can optimize away this check if the basic block does not end with a backwards branch. Consider a program of size N where a basic block that ends with a forwards branch exhausts all remaining gas, and no check is performed. Within at most N instructions, the program must either execute a backwards branch, causing a gas check, or terminate because no instructions remain. Thus, if the basic block exhausts all gas, the program will terminate after at most N further instructions. This means we can cap the value of N at a small amount, such as 10M instructions, and elide all gas checks for forward branches.

4.3 Timer-based Metering

Timer-based metering uses a timer interrupt in combination with the gas counter to enable deterministic preemption. This scheme relies on the insight that a program running in a DeCl sandbox can only have externally visible effects when it makes a runtime call. As long as the program does not make any externally visible changes after it runs out of gas, it does not need to be immediately preempted. Thus, timer-based metering works as follows:

- A timer interrupt is configured to fire frequently.
- When the program makes a runtime call, it is terminated if its gas is negative.
- When a timer interrupt occurs, the program is terminated if its gas is negative.

Since the runtime always checks the gas before applying the effects of a runtime call, it is impossible for a program to have any effect after it runs out of gas, even though it may continue running for a non-deterministic amount of time. To an external observer, the program behaves deterministically. This scheme assumes that a runtime call is the only way to cause an externally visible effect—this would not work if the memory state of the program were also externally visible.

Since gas checks are only performed after interrupts or runtime calls, basic block epilogues only need to update the gas counter, and can always omit the expensive gas check. This makes it possible to use a bundle size of 8 on Arm64. Fig. 4 shows metering sequences for Arm64 and x86-64.

Timer-based metering relies on the ability to configure a timer interrupt, for example via signals on Linux. This scheme also allows a program to use up to t seconds of CPU time after running out of gas, where t is the size of a time slice. While timer-based metering allows all gas checks to be elided, it can cause overhead if the time slice is too short. As a result, branch-based metering may be better for systems that give programs only a tiny amount of gas.

5 Integration with Lightweight Sandboxing

DeCl can be used with hardware-based or software-based memory isolation. For a smart contract engine, fast startup time and context switches are crucial. For this reason, we have integrated DeCl with LFI, a software-based sandboxing system where all sandboxes run in the same address space. LFI supports both x86-64 and Arm64 using techniques described in prior work [40,50,57]. Section 5.1 provides an overview of these techniques, and Section 5.2 discusses the modifications to LFI required to make it deterministic.

5.1 Background: LFI

Lightweight Fault Isolation (LFI) is a sandboxing system that uses software-based fault isolation (SFI). Programs running in LFI are restricted to accessing a 4GiB region of contiguous virtual memory, and may not execute system call instructions or other instructions deemed "unsafe." These restrictions are enforced via a static verifier that analyzes the machine code of untrusted programs. In order to accurately disassemble programs, LFI uses the techniques described in Section 2.2.

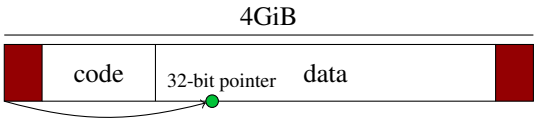


Figure 5: Layout of an LFI sandbox. Red regions indicate unmapped pages.

The basic LFI scheme allocates each sandbox at an address aligned to 4GiB, illustrated in Figure 5. Unmapped guard pages are required between adjacent sandboxes, with sizes determined by architectural details and the chosen sandboxing scheme (typically chosen to be 80KiB for Arm64 and 40GiB for x86-64). Several Arm64/x86-64 registers are reserved for special use and must contain:

- `x21` / `%r14`, `%gs`: the base address of the sandbox.
- `sp` / `%rsp`: an address within the sandbox.
- `x18` (Arm64-only): an address within the sandbox.
- `x30` (Arm64-only): an address within the sandbox.

Note that Arm64 has 32 64-bit integer registers (`x0`-`x30` and `sp`), which can also be accessed via names that only access the bottom 32 bits (`w0`-`w30` and `wsp`). When `wN` is written, the top 32 bits of `xN` are set to zero. Similarly, x86-64 has 16 64-bit registers (`%rax`-`%r15`), which can be accessed via 32-bit subsets (`%eax`-`%r15d`).

All loads and stores are rewritten into safe versions that perform the memory access as a 32-bit offset from the base of the sandbox. This way, the memory access cannot reach memory outside of a 4GiB region, starting from the base address. The following instructions are example of safe memory accesses on Arm64 and x86-64:

```
ldr x0, [x21, w1, uxtw] // Arm64
mov %gs:(%eax), %rdi    // x86-64
```

The first instruction above takes the 32-bit value `w1` (namely the low 32 bits of `x1`), treats it as an unsigned 32-bit value before extending it to 64 bits (specified by the `uxtw` modifier), adds the extended value to register `x21`, loads the value at that computed address, and stores the result in `x0`. Since `x21` contains the 4GiB-aligned base address of the sandbox region, the address that is loaded is guaranteed to be within the sandbox, regardless of the value of `x1`. A similar operation takes place for the x86-64 instruction through the use of `%gs`, an optimization introduced by Segue [40].

We omit many further details of the sandboxing scheme for brevity. Please see prior work on these kinds of sandboxing systems for details [36, 40, 50, 54, 57, 58].

5.2 Position-Oblivious Code

The particular 4GiB slot allocated to an LFI program is determined by the runtime and is non-deterministic, as we want to support running multiple sandboxes at the same time. Since LFI programs are able to read their base address, their stack pointer, or any absolute address used to perform memory accesses, an LFI program can determine where in the address space it is executing, and behave differently depending on this value. While this is not a problem for enforcing memory isolation, it becomes a problem for determinism.

To solve this problem, we introduce *position-oblivious code* (POC): programs that cannot determine their load address. A position-oblivious program may be loaded at any address, and its result is guaranteed to have no dependence on that load address. A program can be verified to be position-oblivious by a static verifier before it is loaded.

Position-oblivious code builds on the LFI technique by statically verifying that absolute addresses are never directly observed. The verifier ensures that reserved registers, which may contain absolute addresses, are only ever accessed via their bottom 32 bits (i.e., `w30` instead of `x30`). Thus, the program may only directly observe offsets from the base address. If the absolute sandbox base address can never be observed, the program is oblivious to where it was loaded.

For example, the following sequence is used to store the return address (`x30`) into the address offset stored in `x0`. This is usually performed as `str x30, [x0]`, but with LFI and position-oblivious code it becomes:

```
mov w22, w30
str x22, [x21, w0, uxtw]
```

This first reads the bottom 32 bits of `w30` into `w22` and then stores that value into memory, instead of the 64-bit `x30`.

Similarly, moves from the stack pointer, such as `mov x0, sp`, are rewritten to read only the 32-bit subset of `sp`:

```
mov w0, wsp
```

Absolute addresses that contain the correct top 32 bits identifying the sandbox location are only stored in reserved registers that are verified to never be directly observed. Before accessing a memory location, the top 32 bits are always set to the correct value and only the bottom 32 bits of a reserved register may be read. On x86-64, the verifier checks that the `%r11` register, which is used for computing safe bundle-aligned indirect branch targets, may only be read as `%r11d` (the bottom 32 bits of `%r11`).

The verifier also ensures that any instructions that produce an address dependent on the program counter (the `adr` and `adrp` instructions, or `%rip-relative addressing`) immediately zero the top 32 bits of the address:

```
adr x0, foo
mov w0, w0
```

These extra constraints mainly affect only three situations: storing the return address on the stack, loading the address of a global, and loading the value of the stack pointer to perform arbitrary computation. These cases are less frequent than regular loads/stores, so the additional overhead is minor.

When implementing a runtime with POC support, care must be taken so that the runtime does not reveal a sandbox's true addresses. For example, pointers returned from runtime calls must have their top 32 bits zeroed. Since POC programs behave the same no matter where they are loaded, they are compatible with a POSIX `fork` API, even though they all exist within the same hardware address space.

x86-64 calls On x86-64, the `call` instruction pushes the return address (an absolute address) onto the stack. As a result, the verifier must reject `call` instructions. The rewriter replaces `call` instructions with a `leal`+`push`+`jmp` sequence that never exposes an absolute address to memory/registers. Unlike a real `call`, this call sequence sets the return address to an instruction further away than the immediate next instruction, and as a result does not need to be placed at the end of a bundle. In some cases this can actually improve performance, as the nops at the end of a bundle after a position-oblivious call sequence are skipped, whereas they would have been executed (and placed before the call) if using a `call` instruction. Returns already must be instrumented to enforce bundling, and will add the base address to the return address before indirect branching. Additionally, instructions like `stos` and `movs` instructions that access memory through `%rdi`/`%rsi` without using addressing modes must zero the top 32 bits of `%rdi`/`%rsi` after executing.

x86-64 flags Finally, information about the sandbox base must not leak via the flags. The `or %r14, %r11` instruction that guards branch targets after the bundle mask sets the ZF, SF, and PF flags. PF is always unaffected by the sandbox base since it only tracks the parity of the bottom byte of the result. SF is unaffected as long as sandboxes are only allocated from one half of the address space (e.g., user addresses), and ZF is unaffected as long as the zero sandbox is never allocated. It is possible to avoid these issues if one wants to allocate sandboxes in both the top (kernel) and bottom (user) halves of the address space by using a slightly larger `lea (%r14, %r11), %r11` instruction instead, which does not affect flags.

5.3 Other Modifications to LFI

Dedicated runtime call register LFI reserves the first page of the sandbox to store metadata about the process, including the entrypoint for runtime calls. This allows the system to reuse `x21` as the address of this metadata page instead of reserving another register for this purpose. However, the contents of this page are not deterministic, so the sandbox cannot be allowed to read it. Thus, DeCl cannot use this page for metadata (DeCl simply leaves it unmapped) or take advantage of the associated optimization, and instead stores the runtime call table outside of the sandbox, while reserving a separate register (`x25`/`%r13`) to contain the address of the runtime call table page. The verifier enforces that this register is only ever accessed for performing runtime calls.

System	Reserved registers	Bundle size
DeCl-HW-x64	`%r11`*	32B
DeCl-LFI-x64	`%gs`, `%r14`, `%r11`, +`%r13`	32B
DeCl-timer-x64	+`%r12`	32B
DeCl-branch-x64	+`%r12`	32B
DeCl-HW-A64	none	none
DeCl-LFI-A64	`x21`, `x18`, `x30`, `x22`*, +`x25`	none
DeCl-timer-A64	+`x24` +`x23`	8B
DeCl-branch-A64	+`x24` +`x23`	16B

Table 3: Register and bundle requirements for all configurations of DeCl, targeting x86-64 (x64) and Arm64 (A64). A + indicates an additional register not reserved by LFI's base configuration. A * indicates a register reserved as a temporary for the rewriter, but its use is ignored by the verifier.

Redundant guard hoisting Guard hoisting is an optimization used by default in the original implementation of LFI, which reserves two registers. To avoid reserving these two additional registers, we use a more limited form of this optimization that simply eliminates redundant guards of the same address without any intervening modifications to `x18`.

Gas register When metering is enabled, DeCl also reserves a register to store gas (`x23`/`%r12`). On Arm64, the `x24` register is also reserved to store bundle-aligned addresses.

6 Implementation

DeCl supports multiple configurations, useful in different scenarios, and summarized in Table 3:

1. DeCl-HW: enforces that programs are deterministic, but relies on hardware protection for memory isolation, and does not provide any metering. This is the configuration that imposes the least overhead, but requires process isolation or a custom kernel to provide memory isolation.
2. DeCl-LFI: provides determinism on top of LFI's memory isolation. This configuration is easy to integrate with Linux applications, since sandbox isolation is handled within a single Linux process. Additionally, sandboxes are extremely lightweight, allowing for faster startup and context switch times compared to DeCl-HW.
3. DeCl-metered: Additional branch or timer-based metering can be enabled in combination with either DeCl-LFI or DeCl-HW. This is necessary for smart contract engines that require preemption. For smart contracts, we choose metered DeCl-LFI for fast startup times.

The implementation of DeCl consists of three components: a compiler, a static verifier, and a runtime implementation. This section describes the first two components. The runtime implementation depends on the application using DeCl. Section 7.2 describes the integration with Groundhog.

6.1 Compiler

Programs are compiled for DeCl using a standard LLVM/GCC toolchain, followed by an assembly rewriter. The rewriter consumes assembly text produced by the compiler and inserts additional instructions. The metering extension inserts sequences before branch instructions. To create aligned bundles, we use the `.bundle` directives supported by Clang and GCC, including `.bundle_align_mode` to set the bundle size, and `.bundle_lock`/`.bundle_unlock` to force sequences of instructions into the same bundle.

6.2 Static Verifier

The DeCl verifier determines if an input is a valid DeCl program. It is a standalone program with limited complexity that runs in linear time. First, it decodes all instructions in the input program and rejects programs that contain any unknown instructions. The decoder is constructed from a fully enumerable set of instructions that only includes well-formed and deterministic instructions. For example, x86-64 instructions with invalid prefixes are never added to the enumerable set, and are thus considered unknown. Instructions from unsupported extensions (e.g., x87 FPU) are also not included.

Next, the verifier ensures that each instruction is safe, depending on the configuration in use. For example, for DeCl-HW-x64, it imposes the following restrictions: (1) No instruction spans a bundle boundary, (2) all direct branches target a bundle-aligned address, (3) indirect calls or jumps are only used as part of a *macro-instruction* consisting of a bundle-masking `AND` immediately followed by the indirect branch.

Finally, on x86-64, the verifier applies the undefined flags analysis described in Section 3.2.2.

When using an LFI-based configuration, the verifier additionally checks for memory isolation properties: loads, stores, and indirect branches are only performed with safe addressing modes/macro-instructions, and reserved registers are only modified in specific ways (to preserve sandbox invariants).

6.2.1 Enforcing Metering

In order to validate metering, the verifier must first determine the program's basic blocks. It does so using the linear-time leader construction algorithm, marking an instruction as a leader if it is the first instruction, any instruction following a branch, or any instruction that is the target of a direct branch.

Next, the verifier iterates through the leaders and enforces that each basic block ends with the appropriate metering sequence. If the basic block ends with a branch, the metering sequence and the branch must be in the same bundle. For example, for DeCl-branch-A64, the metering sequence is:

```
    sub x23, x23, #n  (d1nnn2f7)
    tbz x23, #63, end (b6f80057)
    brk #0            (d4200000)
end:
```

The value `n` is calculated from the location of the current instruction and the previous leader, and is verified against the immediate in the (program-provided) subtraction instruction.

The verifier also tracks the locations of these instructions, and in a final pass ensures that no modifications to `x23` or `tbz x23` instructions occur outside these areas, and that branches do not target instructions within gas update sequences by making sure that branches correctly target aligned bundles.

Some "basic blocks" may be unreachable—for example, the padding between the end of one function and the beginning of another. The assembly rewriter does not know about padding and does not insert gas update sequences there. However, the verifier still expects such padding to be metered. To solve this problem, we allow basic blocks to be merged: if the verifier sees a basic block that ends without a gas update sequence and that has no terminating branch (it just falls through), the verifier accepts the code so long as the next basic block's gas update sequence charges for both blocks. Hence, the first basic block of a function charges for padding between it and the previous function, incurring a small amount of gas overcharging. However, this also means compilers targeting DeCl can opt to merge basic blocks, trading off the possibility of some overcharging for reduced metering overhead.

7 Evaluation

Our evaluation of DeCl seeks to answer the following questions: first, what is the performance overhead of enforcing determinism, metering, and position-oblivious code, and how does this compare with existing state-of-the-art approaches? To answer this, we evaluate DeCl on the SPEC CPU2017 benchmark suite with a variety of configurations.

Next, in the context of smart contracts (the primary industry use-case for adversarial determinism), what tradeoffs does DeCl provide, what metrics are important, and how does DeCl perform on those metrics compared to existing approaches? We evaluate DeCl with Groundhog, a scalable smart contract engine, and evaluate on a payment transaction benchmark (where we find startup overheads to be very important) as well as on contracts that perform zero-knowledge proof verification (where CPU overheads dominate).

7.1 General Performance

To evaluate general CPU overheads imposed by DeCl, we use the SPEC CPU2017 benchmark suite. Benchmarks are

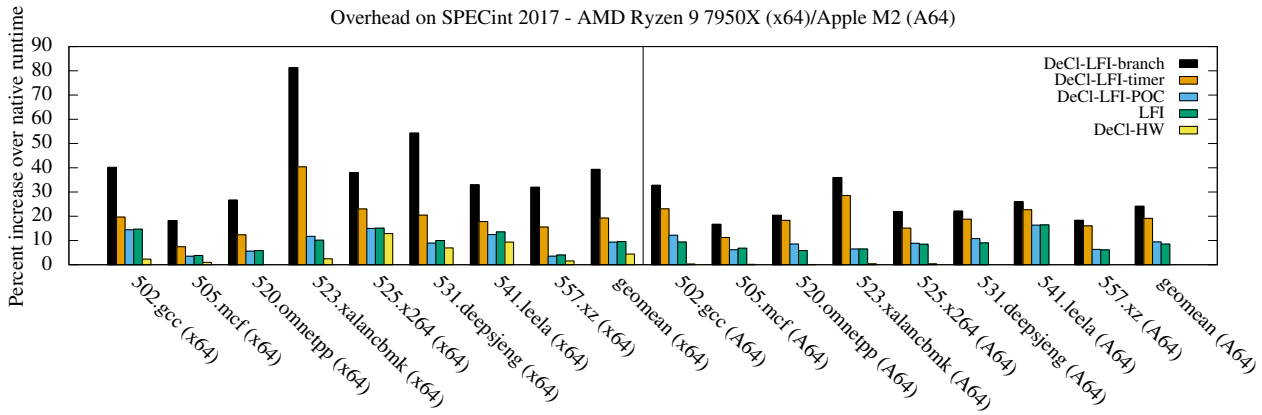


Figure 6: Overheads of all DeCl configurations on integer SPEC benchmarks.

limited to ones that compile with LFI: they must be written in C or C++ and compile with a Musl/LLVM toolchain. We also use the SPECrate benchmarks rather than SPECspeed due to the 4GiB memory restriction imposed by LFI (SPECspeed requires 16GiB RAM per benchmark). We also exclude the SPEC floating point benchmarks, since we consider floating point out of scope.[7] This restricts the suite to 8 benchmarks. All benchmarks are compiled with LLVM 19.1.4, with link-time optimization enabled.

While the programs in SPEC are compiled to be deterministic, in order to run the benchmarks, the runtime must provide non-deterministic functions, such as system calls that return the time. Our runtime for SPEC provides these non-deterministic functions for evaluation purposes. We only run SPEC programs with DeCl to quantify the runtime overheads caused by rewriting. A system that actually uses DeCl for full determinism would implement a custom runtime with a deterministic API and programs would be compiled in a freestanding environment. We evaluate such a system in §7.2.

We evaluate on a Mac Mini M2 running Debian Asahi Linux (not virtualized) for Arm64, and on an AMD Ryzen 9 7950X running Ubuntu for x86-64. Both machines are configured so that benchmarks run at the fixed base clock rate and are shielded from activity outside the benchmarking core, giving highly consistent results.

Fig. 6 shows the overheads of all configurations of DeCl: DeCl-HW, DeCl-LFI-POC, and DeCl-LFI-metered. We also provide LFI overhead as a point of reference.

Table 4 provides a summary of the geomean overheads. The DeCl-LFI-POC performs similarly to unmodified LFI ($\sim$ 9% versus $\sim$ 8% overhead), since the main change is the addition of position-oblivious code, which in general only affects code that saves return addresses or loads globals. Timer-based metering is the most efficient form of metering, with 19% overhead, since it is able to omit all gas checks, but branch-based metering is not far behind at 20-40% overhead. Branch-based metering is more expensive on x86-64, since expensive flags-preserving metering sequences are often needed. We believe there is potential for more optimization here with a more advanced rewriter/compiler. DeCl-HW only incurs overhead on x86-64 ($\sim$ 3%), entirely due to the use of aligned bundles and the need to rewrite return instructions into indirect branches. On Arm64, DeCl-HW incurs no overhead since it does not use any rewrites (Arm programs do not require bundles in order to be analyzed by a verifier).

[7] We believe a large subset of floating point operations are deterministic and could be supported. Evaluation on the full SPEC suite with floating point yielded lower geomean overheads since FP benchmarks utilize less control-flow and have lower pressure on integer registers.

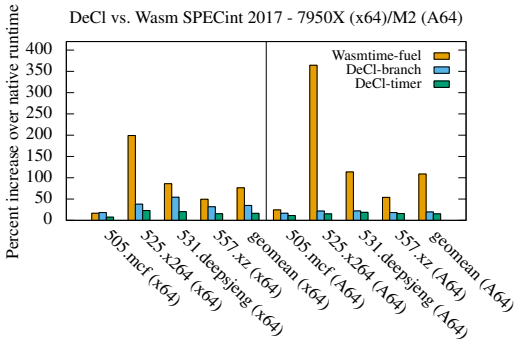


Figure 7: Comparison between metered DeCl and Wasmtime-fuel on integer SPEC benchmarks.

Next we compare DeCl with Wasmtime version 24.0.0 [9], an optimizing WebAssembly JIT compiler designed for running untrusted code, with support for deterministic metering, called fuel. We also enable WebAssembly's 128-bit SIMD extension for Wasmtime. The results are shown in Figure 7. DeCl significantly outperforms Wasmtime in both metered and unmetered configurations, incurring over 2$\times$ less overhead than Wasmtime for equivalent configurations. Table 4 summarizes the geomean experimental results.

System	Fig. 6 x64	Fig. 6 A64	Fig. 7 x64	Fig. 7 A64
Wasmtime-fuel	-	-	76.5%	109%
Wasmtime	-	-	56.3%	82.2%
DeCl-LFI-branch	39.3%	24.1%	35.0%	19.7%
DeCl-LFI-timer	19.2%	19.1%	16.5%	15.3%
DeCl-LFI-POC	9.30%	9.40%	7.64%	8.01%
LFI	9.54%	8.52%	8.11%	7.59%
DeCl-HW	4.40%	-0.14%	5.47%	-0.16%

Table 4: Summary of geomean overheads from Figure 6 (full set of supported benchmarks) and Figure 7 (only benchmarks supported by WebAssembly). Configurations with metering are shown in bold. Some configurations are only shown in this table and not in the referenced figures due to space constraints.

7.2 Application to Smart Contracts

We integrate DeCl within the smart contract engine Groundhog [45], replacing the WebAssembly interpreter (wasm3 [32]) it previously used. This integration required only minimal changes to Groundhog, as both DeCl and the WebAssembly interpreter have approximately the same input-output behavior. Smart contracts are compiled with picolibc [43] and interact with the blockchain's environment (for example, accessing persistent storage, or requesting metadata such as the current block number) via a set of specific functions; DeCl replaces imported WebAssembly functions with runtime calls.

7.2.1 Optimizing Startup and Teardown

A key challenge with the integration is that Groundhog is designed to scale over many CPU cores via concurrent execution of smart contracts. Maintaining this concurrent execution is possible with DeCl because it supports many separate sandboxes in the same process. However, maintaining the scalability requires care when setting up sandboxes, so as to avoid contention on kernel resources. Additionally, smart contracts can be very short programs (running for less than 100μs), so it is imperative that sandbox startup be fast.

In the existing LFI runtime, sandboxes are loaded and mapped into the address space using `mmap`/`mprotect` This approach is not suitable for Groundhog, since it has high startup overheads (multiple system calls), and is not scalable because the use of `mprotect` causes TLB shootdowns.

We solve this problem by preallocating sandboxes with code and data regions. Programs are given 128KiB of read-execute code, and 128KiB of read-write data. These regions are marked with the appropriate protections only the first time a sandbox is used, and subsequently can be reused for future sandboxes without the need for any system calls (only a `memset` to zero the memory).

We use page aliasing (via in-memory files) to map the code region twice: once within the sandbox as read-execute, and separately within the runtime as read-write. This allows the runtime to write to a sandbox's code region without needing to change any memory protections (necessary for loading sandboxes). On Arm64 (but not x86-64), the runtime must then flush the instruction cache on the executable region. With these optimizations, the time to load, execute, and exit from an empty program is 15μs on the M2 processor, and 2μs on the AMD 7950X. Zeroing pages is faster on the M2, but the cache flush dominates the overall time on the M2.

Splitting a contract's code and data into two separate 128KiB regions requires a slight modification to the default linkerscript used by GNU LD. We have chosen sizes for these regions that are as small as possible, while still being usable, because these regions must be cleared after the sandbox terminates. The throughput of the `memset` operation to perform this clear can become a bottleneck if the regions become too large (such as 1MiB). For workloads with a wide variety of contract sizes, the system could pre-allocate sandboxes with varied code/data region sizes.

Code caching As a further optimization, it is possible to cache a sandbox's code and reuse it if the same contract is launched again (skipping the `memset` operation for the code region). This is effective for commonly used contracts like popular ERC20-like tokens, but at the moment our system does not implement this optimization.

7.2.2 Native Cryptography Primitives

One major benefit of DeCl over WebAssembly sandboxes for practical smart contracts is that DeCl enables users to efficiently implement their own cryptography. Even basic operations, like verifying a signature, are sufficiently expensive that today's blockchains must provide hard-coded runtime calls for common operations. This allows the cryptography to run at the speed of native code, bypassing sandbox overhead, but limits the operations available to smart contracts.

Ethereum [55] for example, implemented the BN254 curve [12, 41] within its smart contract environment [15, 46]. Improvements to the cryptanalysis [31] have given this curve less than 128-bit security, and users may wish to use a stronger cryptographic primitive, such as BLS12-381 [11, 13, 49]. Yet applications building on Ethereum have no option to change their cryptographic tools, precisely because deploying a new special-cased operation in an active blockchain requires a difficult, coordinated upgrade. By contrast, DeCl allows contracts to implement their own cryptographic functions internally, and run fast enough that little performance is lost.

Fig. 8 plots the throughput of Groundhog using various sandbox environments: DeCl (branch and timer), Wasm3, and Wasmtime using fuel and the pooling allocator. Each transaction sends a payment between two accounts (of 1,000,000) chosen uniformly at random. Throughput is measured on

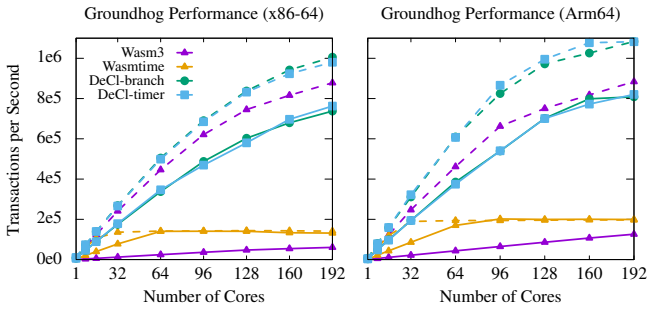


Figure 8: Groundhog throughput on varying numbers of threads, showing that DeCl does not impede scalability. The solid line shows performance when the smart contract provides the implementation of Ed25519, and the dashed line shows performance when Ed25519 is provided by the runtime as unsandboxed native code.

batches of size 100,000. These experiments are run on one c7a.metal (x64) or c8g.metal (Arm64) instance in an Amazon Web Services datacenter. Each system has a 192-core processor (Graviton 4 or AMD EPYC) without hyperthreading.

Each of these transactions verifies one Ed25519 signature and microbenchmarks show that approximately 80 to 90% of each transaction's runtime is spent on that computation. As a result, startup time is critical for this benchmark, since each contract is very short-lived. Wasm3 scales well with access to a native Ed25519 in the runtime thanks to low startup time, but significantly degrades once it must perform cryptography in the sandbox. Wasmtime performs better at low parallelism in both cases but hits a scaling limit due to startup overhead. DeCl can combine low startup time with low CPU overhead, allowing competitive performance even when all of the cryptographic operations run within the sandbox. A smart contract developer can implement alternative cryptographic primitives internally and still attain good performance.

We also evaluate DeCl on zero-knowledge proof verification workloads, which are often some of the most CPU-intense programs run on-chain in smart contracts. We benchmark verification time for Groth16 and Plonk proofs generated using the SP1 zkVM [8] and verified using SP1's `no_std` Rust verifier. Table 5 shows overheads associated with DeCl, Wasmtime (JIT compiler with SIMD enabled), and Wasm3 (interpreter). With DeCl, runtimes for contracts that perform on-chain zero-knowledge proof verification would be reduced by at least a factor of 2, and up to a factor of 30 for blockchains that use an interpreter for contract execution.

8 Limitations

Portability By enforcing determinism at the machine code level, we lose some degree of portability. While the programs can be run on architectures other than their native architecture

System	Groth16	Plonk
Native (x64)	.313s	.588s
DeCl-timer	.344s (1.10×)	.650s (1.11×)
DeCl-branch	.407s (1.30×)	.763s (1.30×)
Wasmtime-fuel	.745s (2.38×)	1.38s (2.34×)
Wasm3	10.5s (33.7×)	20.4s (34.7×)
Native (A64)	.189s	.338s
DeCl-timer	.202s (1.07×)	.365s (1.08×)
DeCl-branch	.210s (1.11×)	.379s (1.12×)
Wasmtime-fuel	.587s (3.11×)	1.08s (3.07×)
Wasm3	5.38s (28.5×)	10.1s (30.0×)

Table 5: Groth16 and Plonk zero-knowledge proof verification time on the 7950X (x64) and M2 (A64) machines. Times are shown along with the slowdown factor relative to native code.

via binary translation, this can come with a significant performance loss, and sacrifices security benefits by introducing a binary translator after verification. Depending on the binary translator, the overhead of translation can be estimated to be around 25%-65% on average for high-performance binary translators [19, 22]. Additionally, a binary translator designed for sandboxing (e.g., using Cranelift as a backend) may have further overheads.

Compiler Toolchain Since the x86-64 and Arm64 instruction sets bundle SIMD and floating point together in the same extension, compilers likewise cannot separate the two. DeCl's verifier allows SIMD instructions while prohibiting floating, which presents challenges with existing compiler toolchains. In particular, it is not possible to enable soft-float support while also keeping SIMD enabled—the compiler only supports enabling/disabling FP/SIMD as a single unit. Currently, we just enable FP/SIMD and catch floating point instructions in the verifier, either rejecting the program, or replacing them with nops. A better solution would be to modify the compiler to allow soft-float and hardware integer SIMD to coexist.

Fragility and Hardware Bugs As part of our threat model, we assume that the hardware within the subset of instructions we allow is trustworthy. While this is likely to hold in practice, it is not necessarily the case. Hardware bugs can occur due to overclocking or slow degradation of the silicon, or can occur due to an error in the hardware design. The former results in glitches that happen sporadically, while the latter is typically restricted to a particular microarchitecture, but deterministically causes a glitch until the hardware vendor releases a fix. Smart contracts already run within byzantine-fault-tolerant systems, and these hardware faults are infrequent and independent enough that as a result they are unlikely to cause issues in practice. Hardware bugs are likely easier to exploit in DeCl than in a JIT-based system, though in many cases they may still be exploitable in both, and are likely much more

difficult/impossible to exploit in an interpreter-based system. Once instruction sequences that trigger hardware bugs are discovered, the verifier can be patched to detect them. If there are existing programs with these sequences, those programs must either be disabled or run in an emulator. Note that since our scheme allows the program to read its own code, the emulator must provide the illusion that the old unpatched code is running, while actually running a different sequence that does not trigger the hardware bug. A different SFI scheme could be chosen to avoid this issue—one which would strictly separate code and data into distinct 4GiB regions.

9 Related Work

Fast & Secure Sandboxing Any replicated state machine that runs untrusted programs requires a sandbox that both runs deterministically and can preempt a contract after an execution limit. Bitcoin [39] achieves deterministic termination with a scripting language without loops [6], while other blockchains run more complex virtual machines like WebAssembly [24], eBPF [37], or the Ethereum Virtual Machine [55]. Our approach is a variant of Software Fault Isolation [36, 50, 54, 57, 58], which instruments native code and verifies that the code cannot escape the sandbox. We build on top of the existing Lightweight Fault Isolation project [57]. A recent proposal for Ethereum suggests using RISC-V machine code as the language of smart contracts [14], which could make use of our technique to allow smart contracts to execute natively on appropriate hardware rather than in an emulator.

These sandbox designs face three key challenges in practical systems, beyond the requirements of deterministic, metered execution. First, they must run as fast as possible; any overhead reduces overall system throughput, leading to higher fees for end-users. Executing smart contracts is a key bottleneck in some blockchains today [25]. Techniques like optimistic concurrency control [23], speculative execution [16], or selective transaction (re)ordering [34, 44, 48, 52, 56] can provide throughput improvements, but these approaches are complementary to faster sandboxing.

Second, any sandbox must execute code securely; any possibility of reading or writing data outside of the sandbox can introduce non-determinism, which can cause two replicas of a state machine to disagree on system state. Translating the bytecode of an IL into efficient machine code is a complex task. Security relies on the correctness of key tools, like the eBPF verifier or a WebAssembly compiler, which are large software systems that have previously contained serious bugs [2, 4, 29]. Approaches include software bytecode interpreters, to minimize complexity (at the cost of runtime overhead), simplifying a JIT compiler [7], formal verification of the compiler [53].

Program Metering Mellor-Crummey et al. implement a native-code instruction counter in software for profiling, using a reserved counter and instrumenting only the backwards branches of a program [38]. Unlike this approach, our metering technique must be verifiable so that the verifier can guarantee that programs are properly metered.

Wasmtime considered a proposal for slacked metering [51], which is similar to our timer-based metering, but it was never fully implemented.

Determinism One example application of determinism is efficiently distributing computation across many machines, as performed by `gg` [21]. Another example is the Exokernel file system, XN [30], which allowed applications to supply their own code for parsing file system data structures as untrusted deterministic functions or UDFs. DeCl provides a more efficient alternative to UDFs.

Record and replay systems [33, 35, 42, 47] can run unmodified, often parallel, programs in a deterministic way. These work by running a program and capturing the results of any non-deterministic operations. Then the program can be deterministically replayed. However, this setting is non-adversarial and does not provide bounded termination or isolation. In a similar vein, Determinator [10] offers a deterministic OS API that can handle parallel programs and is compatible with conventional OS abstractions.

10 Conclusion

We presented Deterministic Client (DeCl), a software sandboxing system that can enforce that sandboxes are deterministic and metered, while running them at near-native speeds. We explain how to guarantee deterministic execution of native code even in an adversarial setting, for the x86-64 and Arm64 architectures. The verification process we describe uses linear-time algorithms and, after some rewrites to the assembly generated by the compiler, can accept programs produced by LLVM. These properties make it possible for DeCl to run smart contracts that are written as machine code programs, enabling bare-metal smart contracts. We demonstrate the feasibility of this approach by integrating DeCl into Groundhog, a scalable smart contract engine.

Acknowledgements

We thank Emmett Witchel, Matthew Sotoudeh, Akshay Srivatsan, and Michael Paper and the anonymous reviewers for their valuable feedback. This material is based upon work supported by the Stanford IOG Research Hub and Future of Digital Currency Initiative, and the National Science Foundation Graduate Research Fellowship under Grant No. DGE-2146755.

References

[1] Mitigations for jump conditional code erratum. https://www.intel.com/content/dam/support/us/en/documents/processors/mitigations-jump-conditional-code-erratum.pdf, November 2019.

[2] CVE-2021-32629. https://www.cve.org/CVERecord?id=CVE-2021-32629, May 2021.

[3] CVE-2023-20593. https://www.cve.org/CVERecord?id=CVE-2023-20593, July 2023.

[4] CVE-2023-26489. https://www.cve.org/CVERecord?id=CVE-2023-26489, March 2023.

[5] Redundant prefix issue. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/redundant-prefix-issue.html, November 2023.

[6] Bitcoin wiki: Script. https://en.bitcoin.it/wiki/Script, 2024.

[7] Cranelift. https://cranelift.dev/, 2024.

[8] Introducing sp1: A performant, 100% open-source, contributor-friendly zkvm. https://blog.succinct.xyz/introducing-sp1/, 2 2024.

[9] Bytecode Alliance. Wasmtime, 2023. URL: https://wasmtime.dev/.

[10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[11] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks: Third International Conference, SCN 2002 Amalfi, Italy, September 11–13, 2002 Revised Papers 3*, pages 257–267. Springer, 2003.

[12] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International workshop on selected areas in cryptography*, pages 319–331. Springer, 2005.

[13] Sean Bowe. Bls12-381: New zk-snark elliptic curve construction. https://electriccoin.co/blog/new-snark-curve/, mar 2017.

[14] Vitalik Buterin. Long-term l1 execution layer proposal: replace the evm with risc-v. https://ethereum-magicians.org/t/long-term-l1-execution-layer-proposal-replace-the-evm-with-risc-v/23617, 2025.

[15] Vitalik Buterin and Christian Reitwiessner. Eip-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. Technical report, Ethereum Improvement Proposals, 2018.

[16] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 570–587, 2021.

[17] Quentin Colombet. Commit 7f4f923, 2014. URL: https://github.com/llvm-mirror/llvm/commit/7f4f923aa57ad8d7ed3f84c532b583f590a68a49.

[18] Alexis Engelke. Fadec — fast decoder for x86-32 and x86-64 and encoder for x86-64, 2024. URL: https://github.com/aengelke/fadec.

[19] Alexis Engelke and Martin Schulz. Instrew: leveraging llvm for high performance dynamic binary instrumentation. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 172–184, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381319.

[20] Chris Fallin. Cranelift: implement "precise store traps" in presence of store-tearing hardware., 2024. URL: https://github.com/bytecodealliance/wasmtime/pull/8221.

[21] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[22] Andrei Frumusanu. The 2020 mac mini unleashed: Putting apple silicon m1 to the test, 2020. URL: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/6.

[23] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.

[24] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062363.

[25] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, and Roger Wattenhofer. Defi and nfts hinder blockchain scalability. In *International Conference on Financial Cryptography and Data Security*, pages 291–309. Springer, 2023.

[26] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2024. Vol. 2A 3-125.

[27] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2024. Vol. 2B 4-639–4-641.

[28] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2024. Vol. 1 A-1–A-4.

[29] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 150–157, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3593856.3595892.

[30] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 52–65, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/268998.266644.

[31] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Annual international cryptology conference*, pages 543–571. Springer, 2016.

[32] Wasm3 Labs. Wasm3, 2024. URL: https://github.com/wasm3/wasm3.

[33] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, April 1987. doi:10.1109/TC.1987.1676929.

[34] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, July 2020. doi:10.14778/3407790.3407808.

[35] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 693–708, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3037697.3037751.

[36] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USA, 2006. USENIX Association.

[37] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[38] John M Mellor-Crummey and Thomas J LeBlanc. A software instruction counter. *ACM SIGARCH Computer Architecture News*, 17(2):78–86, 1989.

[39] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[40] Shravan Narayan, Tal Garfinkel, Evan Johnson, Zachary Yedidia, Yingchen Wang, Andrew Brown, Anjo Vahldiek-Oberwagner, Michael LeMay, Wenyong Huang, Xin Wang, Mingqiu Sun, Dean Tullsen, and Deian Stefan. Segue & colorguard: Optimizing sfi performance and scalability on modern architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 987–1002, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3669940.3707249.

[41] Yasuyuki Nogami, Masataka Akane, Yumi Sakemi, Hidehiro Kato, and Yoshitaka Morikawa. Integer variable χ–based ate pairing. In *International Conference on Pairing-Based Cryptography*, pages 178–191. Springer, 2008.

[42] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan.

[43] Keith Packard. picolibc - a c library designed for embedded 32- and 64- bit systems., 2024. URL: https://github.com/picolibc/picolibc.

[44] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

[45] Geoffrey Ramseyer and David Mazières. Groundhog: Linearly-scalable smart contracting via commutative transaction semantics. *arXiv preprint arXiv:2404.03201*, 2024.

[46] Christian Reitwiessner. Eip-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. Technical report, Ethereum Improvement Proposals, 2018.

[47] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999. doi:10.1145/312203.312214.

[48] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 543–557, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389693.

[49] Yumi Sakemi, Tetsutaro Kobayashi, Tsunekazu Saito, and Riad S. Wahby. Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-11, Internet Engineering Task Force, November 2022. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-pairing-friendly-curves/11/.

[50] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 1–12. USENIX Association, 2010. URL: http://www.usenix.org/events/sec10/tech/full_papers/Sehr.pdf.

[51] Sergei Shulepov. Slacked fuel metering, 2022. URL: https://github.com/bytecodealliance/wasmtime/issues/4109.

[52] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 1–12, 2012.

[53] Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. Lightweight, modular verification for webassembly-to-native instruction selection. ASPLOS, 2024.

[54] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Andrew P. Black and Barbara Liskov, editors, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 203–216. ACM, 1993. doi:10.1145/168619.168635.

[55] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[56] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: a lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.

[57] Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 649–665, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3620665.3640408.

[58] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, 2010. doi:10.1145/1629175.1629203.

[59] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization storage channels (S2C): Timer-less cache Side-Channel attacks on the apple m1 via hardware synchronization instructions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1973–1990, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong.

Disentangling the Dual Role of NIC Receive Rings

Boris Pismenny*
EPFL and NVIDIA

Adam Morrison
Tel Aviv University

Dan Tsafrir
Technion – Israel Institute of Technology

Abstract

CPUs parallelize packet processing across cores via per-core receive (Rx) rings, which are typically sized to absorb bursts with ≥1Ki entries by default. The combined I/O working set (packet buffers pointed to by all Rx rings) easily exceeds the LLC capacity, thus degrading performance due to high memory bandwidth pressure. Recent work has reduced the I/O working set size by sharing Rx rings among cores with the "shRing" system. But this approach suffers from a bottleneck under imbalanced loads, which are common.

We contend that the bottleneck stems from an unnecessary entanglement of two orthogonal producer-consumer structures: (1) memory allocation, where the core produces empty buffers that the NIC consumes to store packets; and (2) packet delivery, where the NIC produces incoming packets that the core consumes. We propose rxBisect, a new CPU-NIC interface that decouples these structures. RxBisect replaces each Rx ring with two separate rings corresponding to the two structures, allowing memory allocation to be performed independently of packet reception. RxBisect can thus pass empty buffers efficiently between cores upon imbalance, thereby eliminating the aforementioned bottleneck. We implement rxBisect with software emulation and find that it improves throughput by up to 20% and 37% relative to the state-of-the-art (shRing) and state-of-the-practice (per-core Rx rings).

1 Introduction

With Ethernet speeds growing to hundreds of gigabits per second (Gbps), the performance of network-intensive applications depends on the effectiveness of technologies such as direct data I/O (DDIO) [15]. DDIO enables the network interface card (NIC) to perform direct memory accesses (DMAs) that read from and write to the last-level cache (LLC) instead of main memory. Software can thus access packets more quickly and conserve main memory bandwidth [1,13,18,23,51,54,64,65,74,79], enabling applications to achieve higher throughput and lower latency.

The effectiveness of DDIO depends on the size of the I/O working set, which consists of the memory regions DMAed by the NIC for packet exchange [66]. If the I/O working set exceeds the LLC capacity, newly arriving packets written by the NIC can evict from the LLC other packets that have not yet been processed [22, 79]. As a result, CPU accesses to packet data slow down because they are served from main memory, which can then become a bottleneck resource [46, 54], degrading both throughput and latency.

The I/O working set is determined by the CPU-NIC packet exchange interface (§2), which consists of two types of producer-consumer queues: one for transmission (Tx) and another for reception (Rx). These queues are circular arrays, known as "rings," whose entries are architected descriptor structures that point to packet buffers. To support high data rates, NICs distribute incoming traffic across multiple per-core Rx rings, enabling systems to sustain ≥ 100 Gbps rates using multicore parallelism [25,27,53,54,59,67,78].

Each Rx ring is initially filled by software with empty 1500 B buffers, matching the network's maximum transmission unit (MTU). As packets arrive, the NIC consumes empty buffers in cyclic order by populating them with incoming packets. Software then consumes the full buffers in the same cyclic order, removing each for processing and immediately replacing it with an empty buffer.

This producer-consumer NIC-CPU interaction implies that every empty buffer in an Rx ring must be used before any can be re-used. Consequently, the I/O working set size is at least $N \times R \times 1500\,\text{B}$, where N is the number of rings and R is the size of each ring. The default ring size for ≥100 Gbps NICs is $R = 1{,}024$ entries [11,16,47,55,73,73,80], required to absorb packet bursts caused by increasing network speeds [2,40,56]. Since CPUs typically provide less than $R \times 1500\,\text{B} \approx 1.5\,\text{MiB}$ of LLC capacity per core, multicore network processing can easily exceed the overall LLC capacity.

Keeping the I/O working set in the LLC thus requires reducing the number or size of Rx rings. Reducing size means rings might be unable to absorb packet bursts, which degrades the throughput a single core can sustain without incurring packet loss [66,83]. We therefore rule this alternative out.

The other alternative is more practical: reducing the number of Rx rings by sharing a single ring among two or more cores, as we proposed in the shRing system [66]. The problem is that shRing requires software synchronization, which adds overhead to packet processing. Moreover, shRing can be effective only when packet processing is roughly computationally balanced across the sharing cores. But if some cores are consistently overloaded while others are not, queueing theory dictates that packet loss is inevitable: overloaded cores impede packet delivery to underloaded cores by monopolizing the shared Rx entries. Accordingly, shRing disables itself upon detecting such situations, falling back on default-size private rings, which can exceed the LLC capacity.

*Work done mostly while at Technion.

structure	*producer*	*consumer*	*elements*
memory allocation	CPU core	NIC	empty buffers
packet reception	NIC	CPU core	full buffers

Table 1: *The canonical Rx interface between the NIC and the CPU entangles two producer-consumer structures in one ring.*

In §3, we explain and demonstrate the I/O working set problem and show that real-world imbalance capable of nullifying the potential benefits of shRing is widespread—although it has been referred to as "pathological" [66].

Our new insight is that the root cause of the problem—the inability to shrink or share Rx rings without the aforementioned undesirable side effects—is the canonical Rx interface, which unnecessarily entangles two orthogonal producer-consumer structures: one for allocating empty buffers, with the CPU core acting as producer and the NIC as consumer; and another for delivering full buffers (housing newly arriving packets), where the roles are reversed as summarized in Table 1. Due to this entanglement, we cannot simply reduce the number of buffers allocated per core without also impairing the core's ability to absorb bursts. Nor can we share allocated buffers among cores without simultaneously forcing them to share (and compete over) packet reception capacity.

To solve this problem, we propose rxBisect, which redesigns the Rx NIC interface to disentangle packet allocation from reception (§4). RxBisect splits the traditional circular Rx array into separate allocation (Ax) and bisected reception (Bx) rings, which are independent and may have different sizes. RxBisect supports cross-core receive buffer sharing: the NIC may consume a buffer from any Ax ring to store an incoming packet, regardless of the destination Bx ring, as long as both rings belong to the same software entity and reside on the same NUMA node. To replenish allocated buffers, the NIC posts a notification to a core's Bx ring whenever it consumes a buffer from that core's Ax ring. Commonly, the Ax and Bx rings of a newly arriving packet belong to the same core, so this notification is included in the new packet's Bx descriptor. When processing this notification, the core places a fresh empty buffer into its Ax ring.

With rxBisect, each core can employ a 1 Ki Bx ring, which may be empty or full if, respectively, none or most of the incoming traffic is directed at that core. Allocation rings can then be smaller, reducing the overall I/O working set and achieving the desired effect. Fewer Ax buffers are needed because rxBisect implements cross-core buffer sharing, quickly moving Ax buffers between cores and using them to populate Bx rings as needed, even when a buffer's source Ax and destination Bx are associated with different cores. Notably, rxBisect frees software from synchronization overheads associated with cross-core buffer sharing (which shRing suffers from), offloading them to the NIC.

We prototype rxBisect using emulation on top of a software NIC framework we developed, which also emulates other NIC models: the private per-core ring baseline and shRing. These allow us to compare the performance of emulated and non-emulated versions and assess the emulation fidelity (§5). We show that emulated performance is similar to real performance or worse, reducing throughput by up to 12% and increasing latency by up to 94%. We therefore gain confidence in experimental results that compare emulated rxBisect performance to non-emulated baseline and shRing performance and find rxBisect to be preferable.

Our evaluation setup employs two 100 Gbps NICs. We show that rxBisect improves throughput by up to 37% and reduces packet latency by up to 11x compared to the default-sized per-core Rx baseline. (The significant latency gains occur when rxBisect sustains line-rate whereas the baseline is unable to.) Under load imbalance, rxBisect improves throughput by up to 20% relative to an idealized shRing (§6).

2 The NIC-CPU Interface Today

Software interacts with NICs via per-core logically cyclic arrays called receive (Rx) and transmit (Tx) rings. We focus on Rx rings as they dictate the I/O working set (see §3). The NIC spreads incoming traffic among cores using receive side scaling (RSS [59]). With RSS, when a packet arrives from the network, the NIC selects its destination Rx ring according to a hash computed over packet header fields.

Rx rings combine two producer-consumer functionalities: (1) software producing empty buffers for the NIC to consume by storing incoming packets (memory allocation) and (2) the NIC producing incoming packets for software to consume (packet reception).

Software configures the Rx ring size and allocates it in main memory, prepopulating ring entries, called *descriptors*, with pointers to MTU-sized buffers. When a packet arrives, the NIC writes it to the buffer pointed to by the *head* ("next empty") descriptor index and advances the head unless it reaches the *tail* ("next full") descriptor index. Software processes packets in order, swapping the buffer pointed to by the tail (with the received packet) for a new empty buffer and incrementing the tail unless it reaches the head.

Software informs the NIC about new free buffers (ring tail advances) by means of an MMIO write to a NIC register, known as "ringing a doorbell." In contrast, software does not poll the ring head to detect new packets, as such polling would result in cache line bounces between the NIC and the CPU (if the head were stored in memory) or expensive MMIO reads (if it were stored in a NIC register). Instead, the NIC informs software of produced packets by means of a cache-friendly memory-based protocol, described next.

Completion Rings Modern NICs notify software of delivered packets via per-core in-memory completion ring (CR) structures [24, 35, 57]. (Intel NICs have used CRs since 2022 [30].) CRs, like descriptor rings, are circular buffers. Each CR is associated with one or more descriptor rings. CR

entries indicate which Rx ring descriptors are ready for software processing by specifying their ring and index.

To optimize cache coherence traffic, the NIC exclusively writes CR entries while software only reads them, coordinating access via a sense reverse mechanism [60]. Both the NIC and software maintain a "generation" bit that flips on every pass through the ring. Each CR entry has a "done" bit that the NIC fills with its "generation" bit when a new packet arrives. Software polls the head CR entry, comparing its "done" bit with its internal "generation" bit to detect new packets.

Figure 1 depicts packet reception with CRs. Initially (Figure 1a), three packets arrive for a core whose Rx ring and CR are empty. Packet delivery (Figure 1b) consists of (1) the NIC using RSS to find the Rx ring and CR matching the packet; (2) writing the packet to an Rx descriptor, and (3) writing a CR entry, indicating the index of the Rx descriptor that holds the packet. These entries have "done" set to 1, because the CR's generation is 1. Packet processing (Figure 1c) occurs when the core, polling the head CR entry, notices its "done" flag has changed. It starts traversing "done" CR entries, processing packets pointed to by the Rx descriptors indicated by them, replenishing these buffers, and advancing both rings' tails. Once the core reaches a CR entry with "done" set to 0, it stops and resumes polling the CR.

We remark that a separate CR is used instead of piggybacking this protocol on Rx descriptors to avoid having both the NIC and software writing to the same cache line concurrently (by updating different descriptors in the same cache line), which creates cache contention and hazards [72].

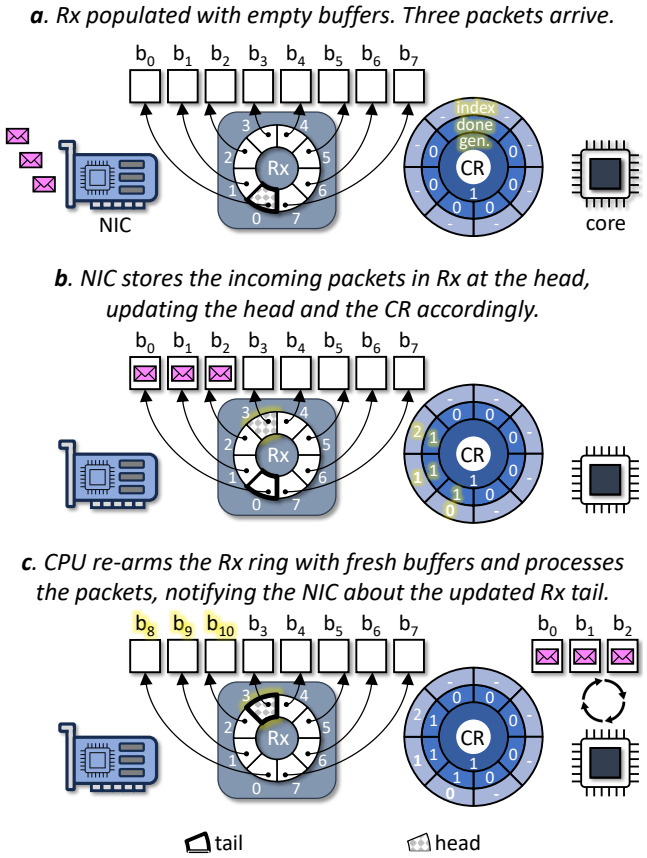


Figure 1: *PrivRing packet reception using a single private receive ring and associated completion ring. Highlighted regions indicates changes relative to the previous stage. For simplicity, we depict only the Rx ring's head and tail positions, assuming identical head/tail positions in the CR (which is not necessarily the case).*

3 Motivation

We focus on kernel-bypass applications that interact directly with the NIC using, e.g., the data plane development kit (DPDK) [38]. These applications process packets in a run-to-completion manner and employ polling to receive packets [3, 39, 43]. The I/O working set problem has been shown to affect socket-based applications as well [66, §6]

3.1 The I/O Working Set Problem

Network-intensive applications depend on DDIO [15] and similar technologies to keep up with network rates of hundreds of Gbps and to enable low-latency packet processing. DDIO allows I/O devices to perform DMAs directly to/from the CPU's LLC, bypassing main memory when possible. With DDIO, DMA reads are satisfied by the LLC if the requested bytes are present. DMA writes place data into the LLC, either overwriting existing cached contents of the receive buffer or allocating new cache lines for it. By default, DDIO may allocate new cache lines in up to 10% of the LLC ways (two ways in the setup described in §6).

When DDIO allocates new cache lines, it evicts existing entries, writing them back to main memory. If this evicted data is later needed for further processing or transmission, it must be fetched from memory. At high data rates, the resulting increase in memory traffic—from both evictions and re-accesses—can create memory bandwidth bottlenecks.

DDIO effectiveness depends on the size of the I/O working set, defined as the memory regions DMAed by an I/O device over some time interval [66]. When an I/O-intensive workload has a working set that exceeds LLC capacity, it leads to the "leaky DMA" problem [22, 79]: new packets written by the NIC evict not-yet-processed packets from the LLC. As a result, CPU accesses to packet data are served from main memory, which is slower and may even become a bottleneck [1, 13, 21, 23, 42, 51, 54, 62, 64, 65, 69, 74, 79, 82]. If slower memory accesses cause a core to fall behind the packet arrival rate, its Rx ring fills up, and latency becomes proportional to the ring size, as each new packet must wait for an entire ring's worth of packets to be processed.

Ideally, the I/O working set would depend only on software processing time, i.e., buffers could be reused by the NIC immediately when released by software. However, the NIC

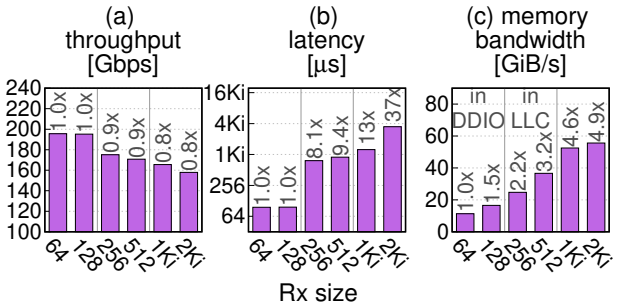


Figure 2: *Large I/O working set causes high memory bandwidth (c), degrading throughput (a) and latency (b). Top labels compare to Rx size of 64. Vertical lines delimit where the I/O working set size fits in DDIO's portion of the LLC, in the LLC, or exceeds the LLC.*

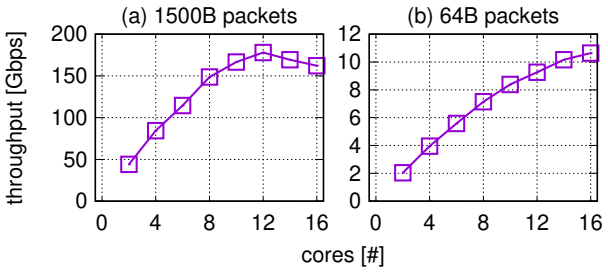


Figure 3: *NF throughput scales with small 64 B packets but not with large 1500 B packets due to the increased I/O working set.*

interface creates a dependency on the number (N) and size (R) of Rx rings. Rx rings are prepopulated with buffers, and as ring descriptors are cyclically accessed, an Rx buffer b can be reused only after the NIC uses all other buffers in the ring, even if software has released b earlier. In contrast, Tx rings contain only in-flight packets, so they are usually empty or partially full. Thus, the I/O working set size is at least the union of all Rx buffers, which is of size $|Rx| = N \times R \times 1500\,\text{B}$.

The growing gap between stagnant CPU speed and ever-increasing NIC bandwidth results in $|Rx|$ growing with hardware advances, thus exceeding LLC capacity [28, 66, 74]. The reason is that this gap necessitates increasing both the size R and number N of the Rx rings, because: (1) packet bursts experienced by individual cores become bigger and should be absorbed to avoid packet loss [20, 37, 81], and (2) packet processing requires additional cycles, disallowing any single core from driving the NIC to its full capacity [27, 65].

We demonstrate the I/O working set problem by evaluating the impact of increasing the Rx ring size on a stateful load balancer (LB) network function (NF). In each experiment, LB uses all cores of a 16-core CPU, which has a 22 MiB LLC and two 100 Gbps NVIDIA ConnectX-5 NICs, for processing 1500 B packets (§6 details the full experimental setup). Figure 2 shows that enlarging the I/O working set worsens throughput by up to 0.8×, latency by up to 37× (due to rings filling, as explained above), and memory bandwidth by up to 4.9×. Line rate throughput is achieved when the I/O working set fits in the two LLC ways used by DDIO (ring size $R \leq 128$). Results degrade in two steps: when the I/O working set exceeds the DDIO ways but fits in the LLC ($128 < R < 1024$) and when it exceeds the LLC ($R \geq 1024$). Other NF applications behave similarly (not shown).

One may wonder why NFs should use all cores, if the result is an excessive I/O working set. The answer is that systems often do not have fixed workloads, and all cores are necessary to maximize throughput in certain workloads. Figure 3 demonstrates this issue by showing LB throughput under maximal rates of either 1500 B or 64 B packets, as the number of LB cores varies. For 1500 B packets, maximal throughput is reached at 12 cores. But for 64 B packets, throughput peaks at 16 cores, as the I/O working remains small. Thus, our goal is to address the I/O working set problem in the most demanding cases—for the benefit of all workloads.

3.2 Limitations of Existing Solutions

We next discuss various approaches for shrinking the I/O working set to fit in the LLC and explain why they are unsatisfactory. In §4.3, after introducing rxBisect, we detail how it avoids the limitations discussed here.

Few Dispatchers This approach, showcased by Shinjuku [42] and Shenango [62], uses a few "dispatcher" cores, each with a large Rx ring, to distribute packets among the remaining worker cores. While such systems can saturate ≈40 GbE links, the dispatcher cores become the bottleneck as link speeds increase to 100 Gbps and beyond [27].

Small Private Rings (PrivRings) A privRing system can employ smaller private per-core rings, such that the I/O working set fits in the LLC. But reducing the Rx ring size below the default (1 Ki) makes rings unable to absorb packet bursts, resulting in packet loss that degrades the performance of loss sensitive protocols such as TCP [66].

ShRing ShRing reduces the I/O working set by sharing default-sized Rx rings among multiple cores [66].[1] ShRing associates several per-core CRs with a single Rx ring, enabling the NIC to spread packets among the sharing cores in a lockless manner. But shRing still necessitates locking and atomic instructions when cores update the shared Rx ring, which adds overhead to packet processing compared to the lockless privRing design (as we show in §6.2).

A major problem with shRing is that it leads to packet loss under load imbalance, where some application cores are consistently overloaded (receiving packets faster than they can process) while others are not. In such conditions, by queueing theory, the overloaded cores monopolize the shared Rx queue and starve their underloaded counterparts for space.

[1] ShRing is based on Mellanox Receive Memory Pools (RMP) [57], which are insufficient to implement rxBisect; see discussion in §7.

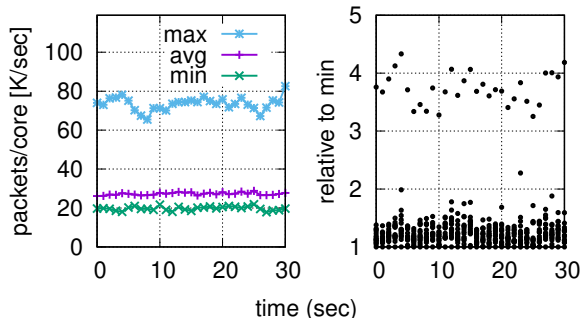


Figure 4: *Imbalanced RSS distribution across 16 cores during a 30-second segment of a real-world packet trace.*

Consequently, shRing is necessarily a dynamic heuristic that should be turned off upon such imbalance.

3.3 Imbalanced Load in the Wild

Imbalance is uncommon in IETF (Internet Engineering Task Force [34]) benchmarks and academic network function (NF) studies. But it does occur in NF workloads [6, 43] and is widespread in other contexts like microsecond-scale remote procedure calls [17, 42, 45, 50, 67, 77].

Figure 4 exemplifies real-world imbalance by analyzing a 30-second sequence from the 2018-03-15 NYC CAIDA dataset [12]. We compute the per-packet Receive Side Scaling (RSS) [59] hash values that the NIC uses to assign packets to 16 cores, and we display the number of packets per second received by each core (left), along with the ratio between the minimum and maximum per-core packet rates (right). The ratio stays between 325–433% throughout. In §6.2, we evaluate dynamic shRing and rxBisect by replaying this trace, and we show that rxBisect throughput is 20% higher as it accommodates imbalance, whereas shRing turns itself off.

4 RxBisect

RxBisect is a new NIC-CPU interface designed to address the I/O working set problem. RxBisect disentangles the traditional Rx ring's empty buffer allocation and packet reception functionalities, allowing them to be managed independently. RxBisect supports two types of rings: *allocation (Ax) rings*, where a core produces empty buffers for the NIC, and *bisected reception (Bx) rings*, where the NIC produces incoming packets, stored in buffers it consumes from allocation rings.

The crux of rxBisect is that each bisected reception ring r can be associated with *several* allocation rings (of different cores), enabling the NIC to deliver packets to r as long as *some* allocation ring is not empty. This association is not exclusive—multiple Bx rings can be associated with overlapping (or identical) sets of Ax rings. In this way, rxBisect turns the union of the set of empty buffers produced by each core into a *globally shared* resource, co-managed by the NIC and software in a lockless manner. This design allows each core to maintain a large bisected reception ring (for absorbing bursts) without having to independently maintain a large set of empty buffers—it only requires a small allocation ring. Thus, rxBisect ensures that the I/O working set size is kept below LLC capacity.

In the following, we detail rxBisect's receive-side processing in the NIC (§4.1) and by software (§4.2). RxBisect does not modify the Tx side and so we do not discuss it. We then pinpoint how rxBisect addresses the limitations of the privRing and shRing designs (§4.3). Next, we discuss NIC hardware implementation issues (§4.4). Finally, we present rxBisect parameter configuration guidelines (§4.5).

4.1 NIC Side

An rxBisect NIC supports two types of receive-side rings: allocation (Ax) and bisected reception (Bx) rings. Each Ax ring entry points to an empty buffer for the NIC to consume in order to store an arriving packet. Each Bx ring entry holds a descriptor through which the NIC notifies software of packet delivery and/or consumption of an empty buffer. Each Bx ring entry holds a pointer to a received packet and the index of the Ax entry and Ax ring that produced the packet's buffer. Bx entries also hold a "done" flag and a corresponding sense-reverse indication flag, whose purpose is the same as in the CRs of the existing NIC interface described in §2.

Figure 5 depicts rxBisect's flow. Initially (Figure 5a), software allocates memory for allocation and bisected reception rings. Allocation rings are filled with entries pointing to MTU-sized buffers to receive packets and bisected reception ring entries are left empty, as the NIC will overwrite them. Crucially, the number of allocated buffers in each Ax ring can be smaller than the size of the Bx rings, which is the key to reducing the I/O working set size (§4.3).

Software then *associates* each Bx ring r with several Ax rings, indicating to the NIC that buffers from these allocation rings can be used to store packets destined to r. Software also *links* each Ax ring a with some Bx ring r, indicating to the NIC that notifications about buffers consumed from a should be delivered through r. For simplicity, assume for now that software (1) allocates per-core Ax and Bx rings, (2) links a core's Ax ring to its Bx ring, and (3) associates every Bx ring in an application with all the Ax rings of that application. (We discuss another software usage model in §4.2.)

When packets arrive (Figure 5b), an rxBisect NIC maps each incoming packet to a Bx ring exactly as a packet is mapped to an Rx ring in today's NICs, e.g., using RSS. For each packet, the NIC chooses an Ax ring with available buffers according to some policy (e.g., the linked Ax ring or a random non-empty Ax ring if it is empty), and consumes a buffer from that Ax ring to store the packet. In the depicted scenario, a burst of five packets arrives for core 0. The first four packets exhaust its Ax ring, and thus the fifth packet is placed in a buffer allocated from core 1's Ax ring. To deliver each packet,

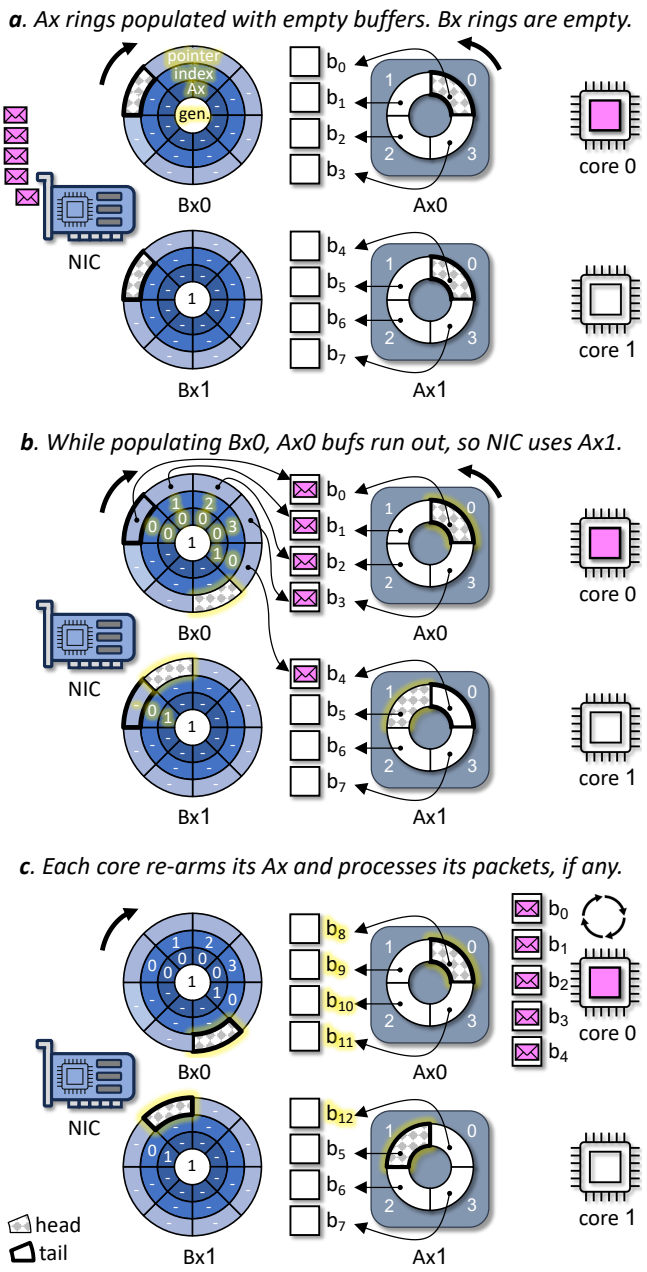


Figure 5: *RxBisect packet reception using two allocation (Ax) rings and two bisected reception (Bx) rings. Ax ring buffers are shared by both Bx rings through NIC hardware. Highlighted text indicates changes relative to the previous stage. The Ax head advances counter-clockwise and the Bx head advances clockwise.*

the NIC first DMA-reads a packet buffer address from the Ax head descriptor entry, and then DMA-writes the packet's data to the packet buffer.

The NIC notifies the receiving core about a delivered packet by DMA-writing the packet buffer's address to the target Bx ring's head descriptor entry. The NIC notifies the allocating core about the consumed buffer by DMA-writing the buffer's Ax ring and entry to the head descriptor of the Bx ring linked to it. In the common case, in which the receiving core is also the allocating core, these notifications are combined into a single descriptor, as shown in Figure 5b for buffers $b_0, \ldots, b_3$. Finally, the NIC updates the Ax and Bx ring head indexes, optionally raising an interrupt for both. Importantly, packet delivery latency that is determined by the number of dependent DMAs, which we call the "critical path," is the same in privRing, shRing, and rxBisect (§4.4).

Finally (Figure 5c), each core processes notifications in its Bx ring (either by polling or following the aforementioned interrupt). It processes delivered packets and/or replenishes buffers consumed from its Ax ring with empty buffers that it allocates, including notifying the NIC (by means of updating the Ax ring's tail) that new buffers are available. After processing a packet, its buffer is freed back to the system allocator. We expand on software-side processing in §4.2.

4.2 Software Side

Our discussion relates to software that directly interacts with the NIC, i.e., kernel-bypass applications or in-kernel drivers. Software has flexibility in how it leverages rxBisect to minimize the I/O working set by configuring Ax rings such that the aggregated size of their buffers does not exceed LLC capacity. For example, software can use small (e.g., 128-entry) per-core Ax rings. Or, software can employ a small number of large Ax rings, which are served by a few dedicated allocation cores while the remaining cores focus on packet reception. The discussion below does not assume a specific configuration. In any case, the buffer-free bisected receive rings should remain large, to absorb bursts. We detail Ax and Bx configuration guidelines in §4.5.

Allocation Mechanism The only constraint rxBisect makes on the software architecture is that it support allocation and freeing of a buffer by different threads/cores. This scenario can occur when a buffer allocated from one core's Ax ring is used to hold a packet destined to a different core, which will then have to free this buffer after processing the packet. Fortunately, many modern multicore allocators support this allocator capability [9]. In particular, both the Linux kernel and DPDK already use such allocators [4, 19]. At a high level, these allocators employ a two-level design consisting of a shared buffer pool with a per-core caching level, which reduces contention on the shared pool. Caches are filled from the shared pool when they run out of buffers and caches drain excess buffers to the shared pool when they grow beyond some threshold of the cache's expected size (e.g., 1.5× in DPDK). The two-level allocator design is important for amortizing the cost of buffer transfer between cores. Due to it, we observe a difference of at most 15 cycles in average allocator call latency between rxBisect and privRing (where buffers never move across cores).

```
1  int RxBisect(Ring *ax, Ring *bx,
2               void **pkts, int len) {
3   uint32_t idx, npkts = 0, nalloc = 0;
4   BXEntry *bxe;
5   while (npkts < len && bxe = consumeBXE(bx)) {
6     if (bxe->buf != NULL)
7       pkts[npkts++] = bxe->buf;
8     if (bxe->idx == ax->idx) {
9       ax->desc[bxe->idx].buf = alloc_buf();
10      nalloc++;
11    }
12  }
13  if (nalloc > 0) {
14    ax->tail += nalloc;
15    *ax->doorbell = ax->tail;
16  }
17  return npkts;
18 }
```

Listing 1: RxBisect disentangled ring receive code.

Receive Flow Listing 1 shows the rxBisect receive function, which dequeues a batch of packets for processing as well as handles notifications about allocated buffers that require replenishing. The function receives an allocation ring (`ax`), a bisected reception ring (`bx`), and an output array of packet pointers (`pkts`) of length `len`. It returns the number of received packets. Multiple cores run this code in parallel with different ring arguments, which are all interlinked by NIC hardware to share Rx buffers.

Lines 5–12 check the Bx ring for new entries. The Bx ring check in `consumeBXE(bx)` uses the sense reverse technique (§2) to identify ready entries without writing to Bx descriptors. When no Bx entry is ready, this function returns `NULL`; otherwise, it returns the first ready entry and updates the Bx ring's tail. (We omit `consumeBXE`'s code.)

Each returned Bx entry indicates both the pointer of a received buffer, if there is one, and an Ax ring and entry index of a buffer consumed by the NIC. If the Bx entry contains a packet buffer, the packet is stored for processing (lines 6–7). If the Bx entry describes a buffer originating from the core's Ax ring, then a new Ax buffer is allocated in its stead (lines 8–10). Lines 13–16 check if allocation requests were handled and advance the Ax ring's tail index if necessary. It is correct to advance the tail because the NIC consumes buffers in ring order and enqueues all the related notifications to this Bx ring in the same order. The code does not limit the number of allocations, to replenish as many buffers as possible. Nevertheless, the number of allocation iterations is bounded by the Ax ring size, because once it becomes empty, only this function can replenish it.

4.3 Comparison to PrivRing and ShRing

By disentangling buffer allocation from reception, rxBisect obtains three advantages: (1) the total number of allocated buffers in Ax rings can be smaller than the total size of the Bx rings, which reduces the I/O working set size; (2) the NIC can use buffers from any Ax ring to populate any Bx ring, so buffer sharing is achieved similarly to shRing; but (3) in contrast to shRing, sharing buffers is achieved without sharing the cores' packet reception capacity or software synchronization.

Figure 6 depicts these differences. The figure compares the minimal I/O working set of two cores running privRing, shRing, and rxBisect. In privRing (Figure 6a), each core can work independently, without synchronization or other dependencies on other cores' behavior. However, the existing NIC ring interface necessitates that an Rx ring able to absorb packet bursts must also hold many empty buffers, resulting in an excessive I/O working set.

ShRing (Figure 6b) solves privRing's I/O working set problem by sharing a default-sized Rx ring, which can absorb packet bursts, between the cores, but it requires synchronization (with locks) to serialize reposting of buffers to the shared ring and advancing the ring's tail. Crucially, shRing always incurs this latency-increasing per-packet overhead, even if the workload does not suffer from the I/O working set problem (e.g., because the packet rate allows a packet's processing to complete before its eviction from the LLC).

In addition, with shRing, traffic destined to an overloaded core can monopolize the shared ring, preventing other cores from receiving packets. E.g., in Figure 6c, because the overloaded core 2 cannot sustain its packet rate, packets destined to it start queueing, eventually filling the ring. Thus, packets to core 1 get dropped, despite it being able to process them.

RxBisect (Figure 6d) combines the advantages of privRing and shRing without inheriting their disadvantages. Disentangling the existing Rx ring functionality into bisected reception and allocation rings with independent sizes allows rxBisect to realize a shared buffer pool, based on small per-core allocation rings—but with cores still working independently, without software synchronization or inter-core dependence. Synchronization is only required when moving freed buffers between cores, but as explained in §4.2, this synchronization is infrequent and its cost is low.

Handling of Imbalanced Load Like shRing, RxBisect reduces the minimal I/O working set by relying on a shared buffer pool, which avoids the over-provisioning of Rx buffers that occurs in privRing. It is therefore natural to ask how rxBisect responds to load imbalance that, in shRing, causes the ring to be monopolized by overloaded cores. As discussed next, thanks to rxBisect's disentanglement of packet reception from buffer allocation, overloaded cores can only "hog" packet buffers, leading to more buffers being allocated, but they cannot interfere with packet reception by other cores.

In rxBisect, per-core Ax rings guarantee the availability of vacant receive buffers for the NIC to consume as long as there is at least one underloaded core with an Ax ring and the buffer allocator can satisfy allocation requests. For allocators preconfigured with a fixed number of buffers, their buffer pool must be large enough to keep satisfying allocation requests even when Bx rings of overloaded cores are full.

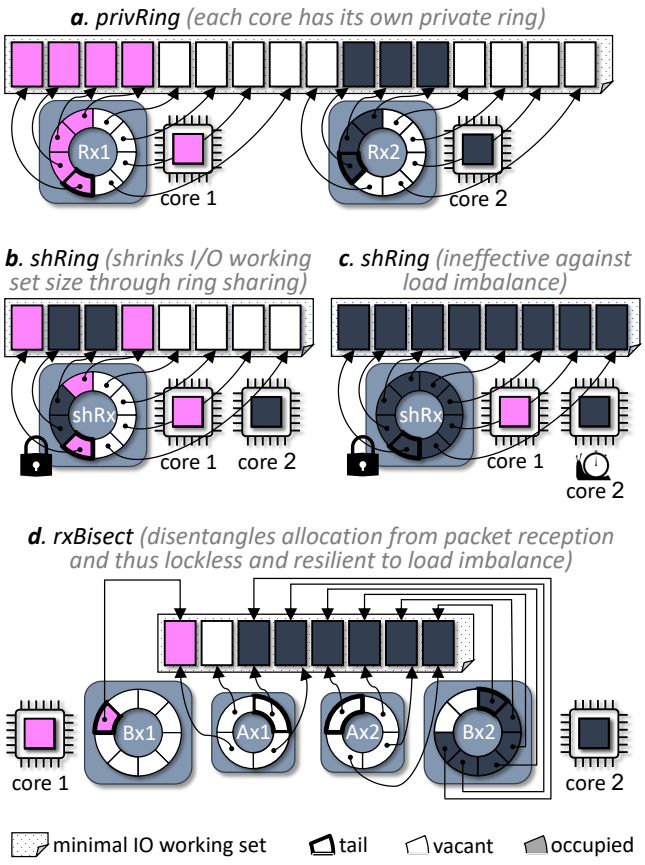


Figure 6: *RxBisect and shRing reduce the I/O working set size compared to privRing. Unlike shRing, rxBisect shares buffers without software locks, and it allows each reception ring to consume buffers from any allocation ring belonging to the same application.*

Let us denote cores 1 and 2 as C_1 and C_2, and consider, for example, the scenario in Figure 6d, where C_2's Ax is exhausted and its Bx is nearly full. Suppose this Bx becomes completely full—which necessarily means it contains buffers from C_1's Ax—and then C_2 stalls. In this case, C_1's Bx contains notifications to repopulate its Ax buffers, and because C_1 is not overloaded, it can process them. Subsequently, C_1's Bx can continue to receive packets using buffers allocated from its own Ax. Meanwhile, the lack of empty Bx entries on C_2 causes the NIC to drop packets of C_2. The important property here is that cores with available cycles *can* receive packets, which is not true of shRing (Figure 6c). Note, however, that the I/O working set might grow due to the additional buffer allocations, beyond the minimal set depicted in the figure.

4.4 Hardware Design Considerations

RxBisect is a new NIC-CPU interface whose complete implementation in ASIC is beyond our scope. Instead, we contrast rxBisect with existing NIC ASICs and show that the necessary changes are compatible with existing NIC mechanisms.

Packet Delivery Mechanically, rxBisect's packet delivery algorithm is analogous to that of existing NICs, which already access two rings (Rx and CR) on packet delivery (§2). Consequently, the critical path of packet delivery in rxBisect is identical to that which current NICs use for privRing and shRing. In all hardware designs, the NIC DMA-reads (and can prefetch) buffer addresses populated by software in ring structures (an Rx ring in current NICs and an Ax ring in rxBisect). RxBisect selects the destination ring in the same way as privRing and shRing when buffers are available, and otherwise it finds a ring with available descriptors based on availability information known from Ax and Bx doorbells. Subsequently, all types of NICs DMA-write packet data followed by DMA-writing a notification in a descriptor of a target ring (a CR in current NICs and a Bx ring in rxBisect). When rxBisect needs to notify different Bx rings about packet delivery and buffer consumption, it performs these DMA-writes in parallel, without increasing the critical path length.

While such parallel writes might increase PCIe bandwidth consumption, this issue can be mitigated by batching notifications. The NIC will delay writing a Bx entry that describes only a buffer consumption until a packet for that Bx ring arrives (or a timeout). Because the waiting Bx entry and the new Bx entry (for packet arrival) are adjacent, they can typically be written with the same PCIe transaction. Similar "completion compression" mechanisms already exist in NVIDIA NICs [32, 33], indicating that this technique is practical.

Hardware vs. Software Synchronization Both rxBisect and shRing implement a shared buffer pool, but shRing places the synchronization burden on software, whereas rxBisect offloads it to hardware as follows. NIC ASICs use pipeline parallelism to meet line-rate speeds. The pipeline stage that assigns packets to ring entries processes them sequentially, one at a time. The rxBisect pipeline stage operates like privRing when both Ax and Bx rings corresponding to an arriving packet have available entries. But it must consume a buffer from another Ax/Bx pair when the destination Ax is empty. This decision can be made efficiently using combinatorial logic circuits, such as priority encoders. We discussed rxBisect applicability to real ASIC NICs with NVIDIA NIC architects, who made the following statement regarding rxBisect's applicability to the ConnectX NIC pipeline [52]:

> *"An rxBisect implementation in ConnectX fits within the existing pipeline stage [that implements shRing] while maintaining current performance for multiple rings with available entries. In the worst case, when a single [Ax] ring provides buffers to several reception rings, performance will be limited by the allocation rate of the single ring."*

4.5 Parameter Configuration Guidelines

We propose the following guidelines to configure Ax and Bx rings. We first discuss the case of a single NIC and subsequently generalize to multiple NICs.

Ax/Bx Association For maximum flexibility, an application should use an all-to-all association of Ax/Bx rings. The reason rxBisect supports configurable Ax-to-Bx associations is to allow different applications (with disjoint Ax/Bx-sets) to work concurrently on disjoint core-sets.

Bx Ring Size ($|Bx|$) The primary consideration in setting $|Bx|$ is that it will be large enough to absorb packet bursts. For 100 Gbps NICs, Figure 3a in the shRing paper [66] shows that $|Bx| = 1024$ is necessary and sufficient. This size is also the default for many systems and NIC vendors.

Ax Ring Size ($|Ax|$) The Ax ring should be sized to achieve two goals: (1) the I/O working set should fit into the LLC's DDIO capacity, i.e., $k \times |Ax| \times 1500\,B \leq |DDIO|$, where k is the total number of Ax rings and $|DDIO|$ is the DDIO capacity; and (2) the Ax rings should have enough buffers to absorb a packet burst without software buffer re-filling, i.e., $k \times |Ax| \geq |Bx|$. In addition, $|Ax|$ cannot be smaller than the vendor-enforced minimum (e.g., 64 in NVIDIA NICs).

In theory, the number of cores may be so large or DDIO capacity so small that $|Ax|$ constraints (1) and (2) cannot both be satisfied, but we did not encounter this issue in practice.

Multiple NICs Cores should be divided among NICs in proportion to traffic demands. For example, if traffic is balanced among the NICs, cores should be equally divided between the NICs. When setting $|Ax|$, the total number of Ax rings over all NICs should be used for k in condition (1) above, but k in condition (2) should remain the number of per-NIC Ax rings.

5 Prototype

Implementing rxBisect requires changes to the NIC ASIC. Therefore, like other related studies [5, 31, 44, 63, 75, 76], we evaluate rxBisect using a software NIC framework, which we developed in DPDK. The framework allows us to run unmodified DPDK applications and emulate all NIC architectures of interest for comparison purposes.

Implementation We implement the emulation using a dedicated core, referred to as the emulator, which acts as the NIC for all worker (application) cores. The emulator uses either multiple physical Rx rings when emulating privRing (one per worker; see emulator in Figure 7a), or a single physical Rx ring shared by multiple workers when emulating shRing and rxBisect (Figures 7b and 7c). Multiple physical Rx rings are required for privRing to ensure strict separation of packet buffers between workers, as placing empty buffers from different workers in the same Rx ring would allow incoming packets for one worker to be stored in buffers allocated by another—violating privRing's isolation property.

The emulator mediates incoming traffic by exposing virtual rings to worker cores (queues on top of the emulator in Figures 7a–7c). It does so by executing an infinite loop that: (1) reads a batch of packet pointers from the physical Rx ring(s); (2) dispatches packets to the appropriate virtual queues (completion rings for privRing and shRing, Bx rings for rxBisect); and (3) replenishes the physical ring(s) with buffers taken from virtual Rx or Ax rings. Dispatching determines the target virtual queues by the least significant bits of RSS hash values, which are computed from each packet's 5-tuple. For privRing, the value is always the same per physical Rx ring, which corresponds to a single virtual ring. In contrast, for shRing and rxBisect, packets from the same physical Rx ring may yield different RSS values.

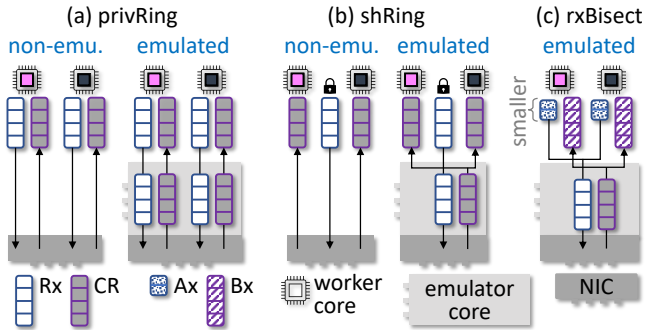


Figure 7: *An emulated NIC uses twice as many Rx entries as the corresponding non-emulated architecture, thereby doubling the I/O working set size. This also applies to rxBisect (as Σ |Ax| = |Rx| in Figure c), so comparing rxBisect to the emulated versions is fairer.*

Usage The I/O working set of each emulated version is twice that of its non-emulated counterpart because: (1) it doubles the number of Rx entries by maintaining a virtual Rx ring in addition to using a physical one; (2) each physical or virtual Rx ring is fully populated by design; and (3) each Rx entry must be used before it can be reused. The 2× ratio also applies to rxBisect, as the combined size of all Ax rings is set equal to that of the physical Rx ring used by the emulator—a ring that would not exist in a hardware implementation.

Thus, methodologically, it is fairer to compare rxBisect to its emulated counterparts, which better isolates performance differences due to changing I/O working sets, constituting an apples-to-apples comparison. Frequently, however, emulated rxBisect performs similarly to or better than non-emulated versions, in which case we prefer to compare it to the latter. These results likely underestimate the benefit of rxBisect.

DDIO DDIO effects (§3.1) are reflected in the emulated setup. To avoid competing with worker cores, the emulator runs on a separate CPU located on a different NUMA node than the workers (Figure 8). DDIO, however, operates only within the node directly connected to the physical NIC (denoted N0). Nevertheless, running the emulation code on another node, remote from the NIC, is not problematic in our framework for two reasons. First, the emulator only manipulates *pointers* to packet buffers and never accesses the packet contents. As a result, packets remain on N0—local to the NIC—ensuring DDIO functions normally from the workers'

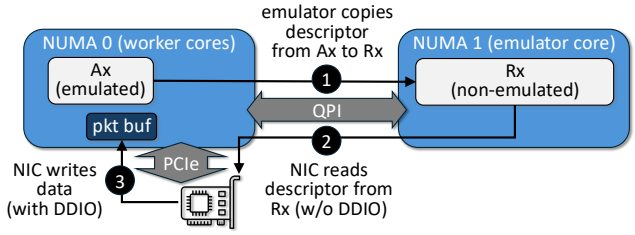


Figure 8: *Emulation captures DDIO effects as packet buffers and worker cores reside on the same NUMA node as the physical NIC.*

		throughput (Gbps)			*latency (µsec)*		
NF	*setup*	*non*	*emu*	***dif***	*non*	*emu*	***dif***
LB	privRing	162	145	**-11%**	1,224	1,432	**+17%**
	small privRing	195	192	**-2%**	110	96	**+15%**
	shRing	194	195	**-0%**	69	69	**+0%**
NAT	privRing	155	136	**-12%**	1,256	2,431	**+94%**
	small privRing	196	179	**-9%**	89	124	**+39%**
	shRing	195	195	**-0%**	67	67	**-0%**

Table 2: *Emulation fidelity (non, emu, and dif stand for non-emulated, emulated, and difference, respectively).*

perspective with respect to packet data. Second, our evaluation focuses on larger ($\leq$ 1500 B) packets to stress the memory subsystem. Unlike packets, the virtual queue descriptors that point to them ping-pong between nodes. But the size of each descriptor (one cache line) is small compared to the associated packet, so the effect is limited.

Overheads To faithfully emulate the overhead of hardware NIC packet delivery, we emulate worker doorbell writes via MMIO writes to NIC memory. Additionally, we emulate the extra DMA write to the Bx ring that occurs when a packet's buffer does not originate from the receiving core's Bx ring by issuing 64 B self-target RDMA writes from NIC memory to host memory.

Fidelity To be useful, the emulation should underestimate the performance achievable with a real hardware NIC, while remaining comparable. We assess whether this is the case in Table 2, which compares the performance of the emulated and non-emulated versions of default privRing, small privRing, and shRing (described in §3.2) using the NAT and LB NFs. Full experimental details are provided in §6. We observe that emulated throughput and latency indeed underperform by up to 12% and 94%, respectively. Emulation latency nearly doubles in the NAT/privRing case as workers are overloaded and each packet must wait through two full Rx rings (virtual and physical) before being processed. Throughput, in contrast, is insensitive to such queueing delays.

6 Evaluation

We evaluate rxBisect using network function benchmarks and the MICA key-value store [49], when load is balanced and shRing performs well (§6.1) and when it does not (§6.2).

System Setup We use two Dell PowerEdge R640 servers. One server is the system under test and the other is the load generator. The load generator runs the stateless Cisco T-Rex packet generator [14], modified to improve latency measurement accuracy [68]. Each server has dual 2.1 GHz Xeon Silver 4216 CPUs, each with 16 cores and an 11-way 22 MiB LLC, and 128 GiB (=4 $\times$ 16 GiB) 2933 MHz DDR4 memory. The servers are connected back-to-back via two pairs of 100 Gbps NVIDIA ConnectX-5 Ethernet NICs [58] and are configured following NVIDIA's DPDK best performance guidelines [61].

All experiments use the default system and application settings. We report trimmed means of ten runs, i.e., with the minimum and maximum discarded. Standard deviation is always below 5%. Throughput results reflect the traffic rate sent back to the load generator after processing, i.e., they discount packets dropped by the server.

Applications We use address translation (NAT) and load balancing (LB) NFs, implemented with FastClick [7]. NAT remaps network addresses. LB assigns flows to one of 32 servers. Both rewrite packet IP headers, using a 10 M-entry hash table to maintain state. As a macrobenchmark, we use the MICA key-value store [49]. Applications run on one CPU of the test server (the other CPU is reserved for emulation). Unless noted otherwise, we dedicate 8 cores to each NIC.

Evaluated Architectures We compare five architectures: (1) rxBisect, with 128-entry Ax ring and 1 Ki-entry Bx ring for each core (emulated); (2) privRing, with the default 1 Ki-entry Rx ring per core; (3) small privRing, which decreases the per-core ring size by 8$\times$ (from 1 Ki to 128) to obtain the same I/O working set size as rxBisect and shRing (proven impractical, as it impedes burst absorption, but included as a yardstick); (4) shRing, with two shared default-sized (1 Ki-entry) Rx rings, one per NIC, such that each is shared by 8 cores (implemented using the RxArr variant [66]); and, when relevant, (5) an idealized version of dynamic shRing, which does not run but instead reports the better result between privRing and shRing.

Comparison Methodology As explained in §5, rxBisect should be compared to the emulated variants. We nevertheless compare it to the non-emulated executions, showing the emulated counterparts (with patterned bars or dashed lines) only when non-emulated execution outperforms rxBisect.

6.1 Balanced Load

This section shows that rxBisect, like shRing, performs well when load is balanced or bursty and it outperforms privRing even when rxBisect is emulated and privRing runs natively.

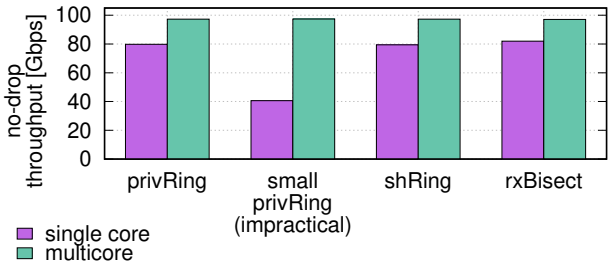


Figure 9: *Buffer sharing allows single-core rxBisect and shRing to achieve the same no-drop throughput as 1 Ki privRing with only 1/8 of its buffers, in contrast to small privRing, which does not share.*

No-Drop Throughput Figure 9 shows the DPDK l3fwd RFC2544 no-drop throughput of one 100 Gbps NIC with 1500 B packets when using an individual core ("single core") and 8 cores combined ("multicore"). RxBisect and shRing's buffer sharing enables them to match privRing's throughput with only 1/8 of its I/O working set, both with multiple cores (each of which handles 1/8 of the traffic) and with a single core, which can absorb bursts by using all available buffers. In contrast, small privRing does not share and thus fails to similarly absorb bursts, which makes it impractical [66].

NAT and LB We run NAT and LB on all 16 cores and load them with 200 Gbps using 1500 B packets. Figure 10 shows (a) throughput, (b) latency, (c) ring occupancy, (d) DDIO hit rate, and (e) memory bandwidth. The latter two are measured using Intel PCM [36]. At this load, architectures with a small I/O working set achieve line rate throughput and comparable average latency (100 µs–119 µs). This is due to their effective use of the 2 LLC ways assigned by default to DDIO (Figure 10e). Since rxBisect is emulated while others run natively, its latency, ring occupancy, memory bandwidth, and DDIO miss rate are higher than the rest (see also §5).

Due to its large I/O working set, privRing throughput collapses by up to 20% compared to rxBisect. As a result of failing to sustain line rate, privRing's Rx rings fill up, causing latency to increase by 11× due to the additional queueing time. PrivRing can achieve line rate in LB if DDIO's LLC portion is increased to 8 LLC ways, but fails to achieve line rate for NAT even if all the LLC is assigned to DDIO. (Of course, exposing more LLC ways to DDIO is a double-edged sword, as I/O and application memory accesses compete [83].)

Key-Value Store We use the MICA key-value store [49] to evaluate the effect of rxBisect beyond NFs. We run MICA on 8 cores (using a single NIC) with 128 B keys and 1024 B values. MICA maps incoming requests to processing cores by hashing the target key. We use workloads with 95% `PUT` requests at the highest possible rate.

Figure 11 shows throughput obtained when the key distribution is (a) uniform or (b) skewed (Zipf with parameter 0.99).

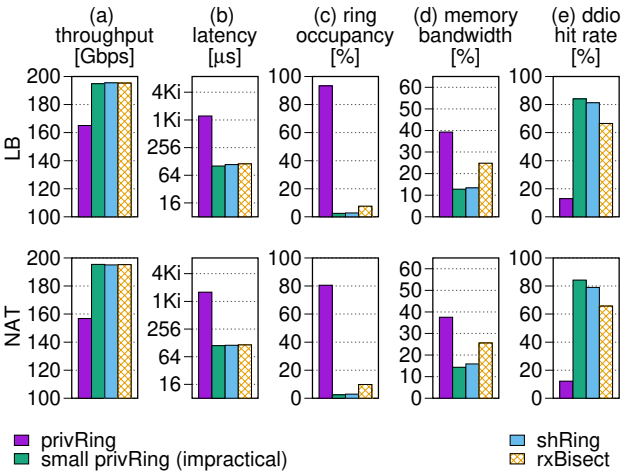


Figure 10: *LB and NAT performance.*

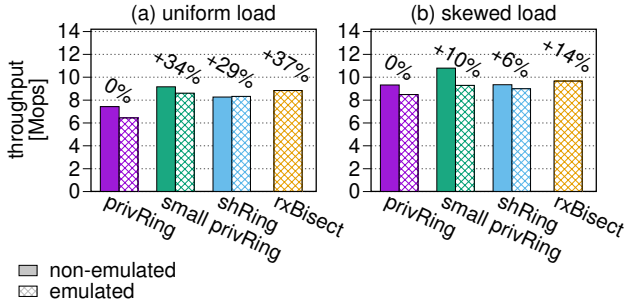


Figure 11: *MICA key-value store throughput with uniform and skewed key distributions. Top labels compare to emulated privRing.*

The throughput of rxBisect is higher than privRing, small privRing, and shRing (all emulated) by up to 37%, 3% and 7%, respectively. The throughput of rxBisect is also higher than non-emulated privRing and shRing by up to 18% and 6%, respectively. RxBisect underperforms (by up to 12%) only the yardstick non-emulated small privRing, which is not a practical architecture.

6.1.1 Impact of Ring Sizes

This section shows the impact of varying Ax and Bx ring sizes. It demonstrates that buffer sharing enables improved burst absorption with small Ax rings, and that the I/O working set is not sensitive to the size of Bx rings.

Varying Ax ring size We evaluate the impact of Ax ring size on no-drop throughput using the RFC2544 [10] benchmark in DPDK l3fwd. The test uses 1500 B packets from a single flow directed at an application running on four cores, while varying the size of rxBisect Ax rings and non-emulated privRing Rx rings. The size of rxBisect Bx rings is fixed at 1 KiB. Figure 12 shows the results.

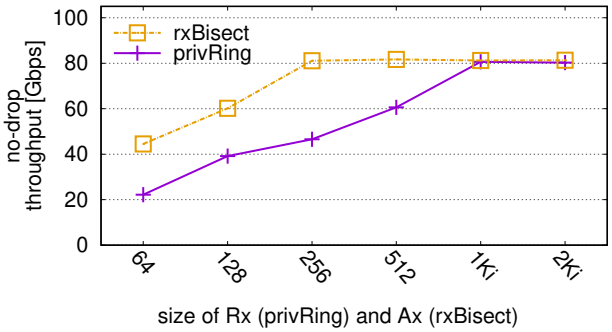


Figure 12: *DPDK l3fwd no-drop throughput, utilizing 4 cores and a single 100 Gbps NIC while varying the size of Ax rings (rxBisect) and Rx rings (privRing). RxBisect Bx rings are set to 1 Ki entries for all runs, similarly to Rx rings. RxBisect buffer sharing improves burst absorption, leading to higher no-drop throughput.*

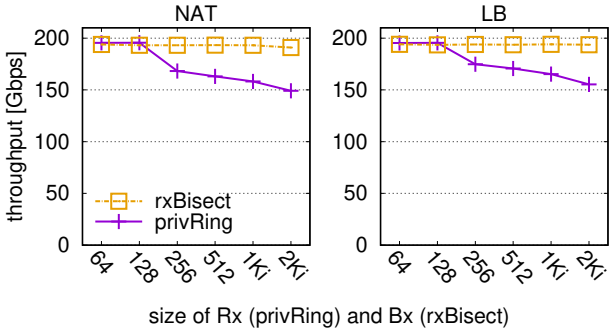


Figure 13: *Throughput at 200 Gbps load while varying the size of Bx rings (rxBisect) and Rx rings (privRing). The size of rxBisect Ax rings is 128 entries for all runs. The rxBisect I/O working set size is constant and small, regardless of Bx size, avoiding the performance degradation experienced by privRing with ≥256 Rx entries.*

RxBisect leverages buffer sharing across the four Ax rings, allowing a single core's Bx ring to absorb bursts up to four times larger than a single Ax ring and up to the size of the Bx ring. Consequently, rxBisect achieves the maximum single-flow no-drop throughput of ≈80 Gbps with just 256 Ax ring entries, whereas privRing requires four times more—1 Ki entries—to reach the same performance.

Varying Bx ring size We evaluate the sensitivity of the I/O working set to Bx ring size using the following experiment. NAT and LB run on all 16 cores under a 200 Gbps load with 1500 B packets. Figure 13 shows throughput as we vary the size of rxBisect Bx rings and non-emulated privRing Rx rings, with rxBisect Ax rings fixed at 128 entries.

As expected, increasing privRing Rx ring size grows the I/O working set, reducing throughput—first as it exceeds the DDIO-allocated portion of the LLC (at 256 entries), and more significantly when it exceeds the total LLC capacity (at 1 Ki entries). In contrast, increasing rxBisect Bx ring size does not increase the working set, sustaining line-rate throughput.

6.2 Unbalanced Load: ShRing Limitations

Next, we show that rxBisect remains effective even when shRing is not. ShRing is ineffective upon imbalance that leads to overload, which may occur because incoming traffic is spread unevenly across cores, or because per-packet processing times are highly-skewed. Additionally, shRing is ineffective when its smaller I/O working set does not compensate for its synchronization overheads.

The dynamic shRing variant addresses these limitations by switching between shRing and privRing at run time using a heuristic that assesses which is currently more beneficial [66]. We therefore compare to this variant as well.

Conceivably, dynamic core frequency scaling can mitigate imbalance by helping overloaded cores at the expense of less busy cores. But when using DPDK, dynamic core scaling is irrelevant due to core busy polling. We confirm this by running with and without scaling and observing similar results.

Processing Variability We evaluate a synthetic NF workload running on all 16 cores and processing 1500 B packets. Packet processing consists of accessing two random addresses in a 40 MiB buffer, performing a routing table lookup (similarly to the l3fwd NF), and sending the packet out. We designate one core per each NIC as the "target core." We modify the load generator to send only 1 Gbps of traffic to the target core while the rest is spread between the other cores. We also tweak the target core's packet processing routine to access memory a configurable number of times per packet (other cores' processing remains unchanged).

Figure 14 presents the resulting throughput. RxBisect and small privRing attain line rate throughput. However, shRing's throughput degrades by up to 60% as the target core's processing slows down, clogging the shared ring with its traffic and blocking other cores from receiving packets (which thus get dropped). Dynamic shRing only offers the best of shRing and privRing, and thus it declines from shRing's line rate speed to privRing's throughput (never above 178 Gbps, due to its I/O working set) when the target core's processing exceeds 100 memory accesses per packet. Consequently, rxBisect outperforms dynamic shRing by up to 12%.

Traffic Variability We use the same synthetic NF as above. Only now, all cores run the same packet processing logic, but we modify the load generator configuration to vary the percentage of packets directed at the target core. We use 64 B packets for traffic directed at the target core and 1500 B packets for traffic directed at all other cores, to minimize the target core's impact on overall throughput.

Figure 15 shows the resulting throughput. RxBisect and small privRing attain line rate throughput. RxBisect experiences a 6% throughput degradation when the target core's

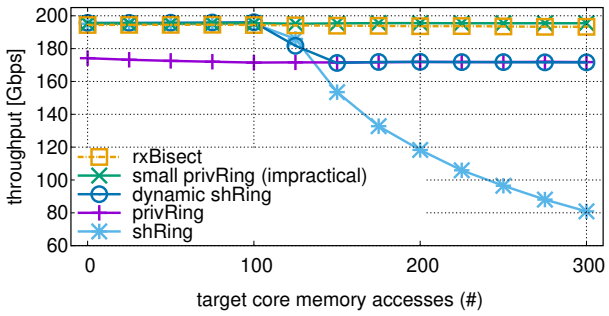


Figure 14: *Processing variability caused by increased memory accesses has no effect on rxBisect throughput despite ring sharing, whereas shRing's sharing degrades throughput when faced with such increased variability.*

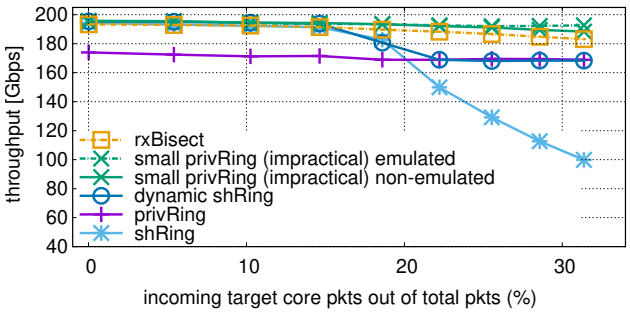


Figure 15: *Traffic variability caused by increased packet arrival rate on a single core has no effect on rxBisect throughput despite ring sharing. In contrast, shRing's approach to ring sharing degrades throughput in this case.*

$\frac{tgt}{total}$ *packets*	*tgt* $\frac{alloc}{total}$	*tgt* $\frac{free}{total}$	*non-tgt* $\frac{alloc}{total}$	*non-tgt* $\frac{free}{total}$
1% (minimum)	0.140%	0.000%	0.002%	0.003%
31% (maximum)	0.001%	0.010%	0.039%	0.000%

Table 3: *RxBisect inter-core memory allocator cycles out of all cycles on target (tgt) and non-target (non-tgt) cores during minimum (1%) and maximum traffic imbalance (31%) to target core.*

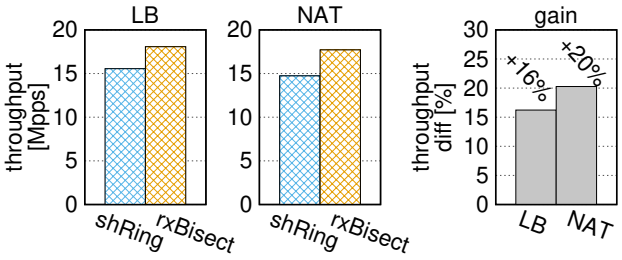


Figure 16: *RxBisect outperforms dynamic shRing when processing the imbalanced CAIDA trace (see §3.3) with PageRank co-location.*

packet rate exceeds 20%, as a result of the emulator core becoming a bottleneck. To demonstrate this, we also show small privRing emulation, whose performance degrades at a rate similar to rxBisect when the target core's packet rate exceeds 25%. ShRing throughput declines by up to 49% as the target core's incoming packet rate increases, again due to its traffic clogging the shared ring. PrivRing throughput never exceeds 175 Gbps, and so dynamic shRing is outperformed by rxBisect.

Inter-Core Memory Allocator RxBisect depends on inter-core memory allocators (e.g., DPDK's `pktmbuf_pool`), which use per-core caches to reduce synchronization. However, rxBisect packet buffer sharing can transfer buffers between cores, potentially stressing the memory allocator (allocating on one core and releasing on another core can deplete the per-core cache on the former).

We analyze the global allocator cycles from the traffic variability experiment. Table 3 presents results for rxBisect at the extremes of imbalance. For other architectures, allocator cycles are always under 0.001% of total cycles (not shown). Overall, the allocator uses less than 0.2% of total cycles in all cases. At minimal target core load, it processes fewer packets, spending 0.14% of cycles in the allocator providing buffers to other cores. At maximal load, non-target cores spend 0.039% cycles allocating buffers to be used by the target core, which spends 0.01% cycles' returning buffers to the global allocator.

Real Packet Trace Figure 16 shows NAT and LB throughput when processing the CAIDA trace analyzed in §3.3, co-located with PageRank [8] to increase memory bandwidth pressure. Dynamic shRing selects the better option between privRing and shRing, but both perform suboptimally: the added memory bandwidth pressure degrades privRing, while the imbalanced load (highlighted in Figure 4) degrades shRing. Thus, rxBisect outperforms dynamic shRing throughput for LB and NAT by 16% and 20%, respectively.

Figure 17 shows results from a similar set of experiments, this time with a varying number of co-located STREAM triad [41] instances, which are memory bandwidth intensive. RxBisect throughput exceeds dynamic shRing by up to 16% (emulated) and up to 8% and 13% (non-emulated) for NAT and LB, respectively. We show both emulated and non-emulated results because, unlike other workloads we studied, CAIDA trace processing slows down under emulation. As a result, non-emulated privRing—and therefore non-emulated dynamic shRing—can outperform (emulated) rxBisect by up to 7% in the absence of memory contention.

With no memory bandwidth contention, dynamic shRing defaults to privRing, which performs well with the imbalanced trace. But as contention increases, shRing begins to outperform privRing, prompting the dynamic variant to switch to shRing. In both cases, when making an apples-to-apples comparison of emulated NICs only (by "usage" in §5), rxBisect performs better, as it uses less memory bandwidth and is resilient to imbalanced load.

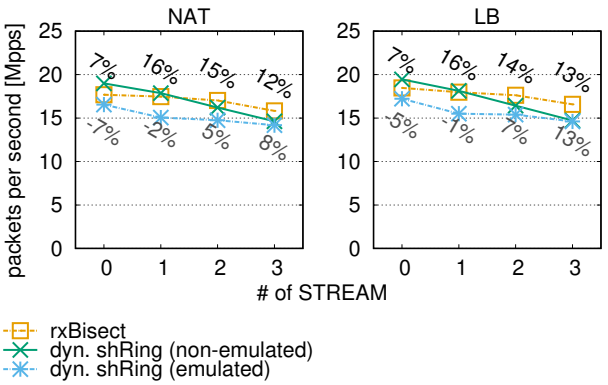


Figure 17: *When comparing emulated setups (apples-to-apples), rxBisect outperforms dynamic shRing regardless of the number of co-located STREAM instances. Top and bottom labels compare rxBisect to emulated and non-emulated dynamic shRing, respectively.*

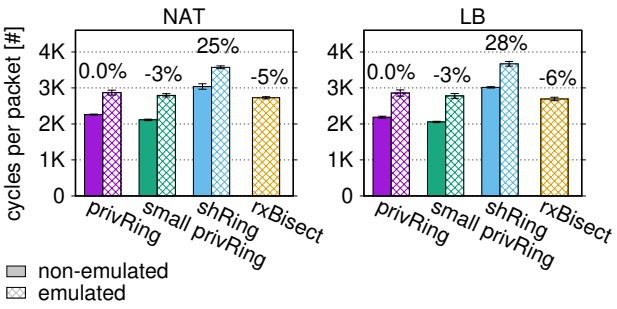


Figure 18: *Under low load, shRing yields no benefit and demonstrates its synchronization overhead. In contrast, rxBisect's overhead is minimal. Top labels compare against emulated privRing.*

Low Traffic Rate Figure 18 shows the per-packet processing time (in CPU cycles) for NAT and LB when arriving traffic consists of 1 Gbps of 1500 B packets, which are spread evenly among all cores. Non-emulated shRing's synchronization overhead results in it spending up to 34% and 46% more cycles per packet than non-emulated privRing and small privRing, respectively. In contrast, rxBisect (which runs under emulation) requires $\approx$ 10% fewer cycles to process a packet than non-emulated shRing. We can safely conclude that with rxBisect hardware, rxBisect will leverage Rx buffer sharing for a small I/O working set with faster packet processing than in shRing. Based on rxBisect's design, which is similar to privRing, and emulation results, we hypothesize that native rxBisect packet processing efficiency will be comparable to privRing's.

7 Related Work

Leaky DMA NICs write directly to two LLC ways with DDIO. Poor I/O working set management causes incoming packets to unnecessarily evict LLC data including other packets being processed; this is also called the "leaky DMA" problem [79]. Several works, complementary to ours, tackled the leaky DMA problem by partitioning the LLC between I/O and applications to avoid interference [83] and by placing only packets headers in the LLC [29, 65, 71].

Host-NIC Interfaces Recently, Ensō [70] proposed a streaming interface for NIC that reduces PCIe overheads which enables line-rate processing of small packets. Ensō is, however, orthogonal to rxBisect because it still suffers from the I/O working set problem which rxBisect solves. The two are complementary: Ensō helps with small I/O working sets and rxBisect helps with large I/O working sets.

Sharing Receive Buffers ShRing [66] and RDMA shared receive queues (SRQ) [35], like rxBisect, share Rx buffers between cores. But both shRing and SRQ perform poorly with load imbalance. Mellanox receive memory pools (RMP) is a hardware abstraction that can be used to share buffers between several Rx rings that reside on the same core [48] or to implement shRing, but RMP still employs a traditional, entangled Rx ring through which both allocation and reception are done. As a result, RMP cannot share buffers between several cores without locking to advance the shared ring's tail pointer. Junction [26] shares receive buffers via RMP and handles load imbalance by using work stealing and a dedicated core that monitors workers and refills ring buffers. RxBisect, in contrast, allows all cores to be workers, improving total packet processing capacity.

Load Imbalance Load imbalance is a fundamental problem in parallel processing. In end-host networking, packets are typically spread among cores using receive side scaling (RSS), which assigns packets to cores based on a hash value computed from packet header fields [59]. Custom RSS functions can rebalance the load between cores [6], but this approach is inherently static and cannot address dynamic load imbalance. Several recent works tackle the dynamic imbalance problem by offloading transport or application layers to the NIC, such that it can rebalance incoming messages among cores according to their load [17, 50, 77]. RxBisect is complementary to these approaches as it helps reduce the I/O working set size by sharing packet buffers without synchronization.

8 Conclusion

Today's NIC interface needlessly overcrowds the LLC with I/O buffers to absorb worst-case packet bursts, degrading the performance of high-throughput applications. RxBisect absorbs bursts with much fewer buffers by untangling the dual role of Rx rings.

References

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo,

Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2022. https://doi.org/10.1145/3563766.3564110.

[2] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/alizadeh.

[3] Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot. Don't share, don't lock: Large-scale software connection tracking with krononat. In *USENIX Annual Technical Conference (ATC)*, pages 453–466, 2018. https://www.usenix.org/conference/atc18/presentation/andre.

[4] Ilias Apalodimas. Page pool API. https://www.kernel.org/doc/html/latest/networking/page_pool.html, 2019. Accessed: 2020-07-14.

[5] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ECN in Multi-Service Multi-Queue data centers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 537–549, 2016. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/bai.

[6] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. RSS++: Load and state-aware receive side scaling. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 318–333, 2019. https://doi.org/10.1145/3359989.3365412.

[7] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16, 2015. https://doi.org/10.1109/ANCS.2015.7110116.

[8] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. *arXiv e-prints*, page arXiv:1508.03619, Aug 2015.

[9] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, 2000.

[10] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, Internet Engineering Task Force, March 1999. http://www.rfc-editor.org/rfc/rfc2544.txt.

[11] Jesse Brandeburg. ice: change default number of receive descriptors. https://marc.info/?l=linux-netdev&m=156771568024262&w=2, 2019. Intel. Accessed: Jun 2021.

[12] CADIA: Center for Applied Internet Data Analysis. The CAIDA UCSD statistical information for the CAIDA anonymized internet traces. https://www.caida.org/catalog/datasets/trace_stats/. Accessed: May 2025.

[13] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 65–77, 2021. https://doi.org/10.1145/3452296.3472888.

[14] Cisco. TRex: Realistic Traffic Generator. https://trex-tgn.cisco.com/. Accessed: May 2021.

[15] Intel Corporation. Intel data direct i/o technology (intel DDIO): A primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012. Accessed: 2020-07-18.

[16] Nithin Dabilpuram. [dpdk-dev] [patch 00/44] marvell CNXK ethdev driver. https://inbox.dpdk.org/dev/20210306153404.10781-4-ndabilpuram@marvell.com/T, 2021. Marvell. Accessed: 2022-11-28.

[17] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of µs-scale rpcs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 35–48, 2019. https://doi.org/10.1145/3297858.3304070.

[18] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154, 2012. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu.

[19] DPDK memory pool library. https://doc.dpdk.org/guides/prog_guide/mempool_lib.html, 2014. Accessed: 2023-08-30.

[20] Matt Faraclas. Received packets have been dropped by NIC. https://indeni.com/blog/cross-vendor-alert-of-the-week-some-received-packets-have-been-dropped-by-nic/, 2014. Accessed: Jun 2021.

[21] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Packetmill: Toward per-core 100-Gbps networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1–17, 2021. https://doi.org/10.1145/3445814.3446724.

[22] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *USENIX Annual Technical Conference (ATC)*, pages 673–689, 2020. https://www.usenix.org/conference/atc20/presentation/farshin.

[23] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the most out of last level cache in intel processors. In *ACM Eurosys*, pages 1–17, 2019. https://doi.org/10.1145/3302424.3303977.

[24] Bailey Forrest. [patch net-next 00/16] gve: Introduce DQO descriptor format. https://lore.kernel.org/netdev/20210624180632.3659809-1-bcf@google.com/, 2021. Google Virtual Ethernet NIC. Accessed: 2022-11-28.

[25] FreeBSD. Network RSS. https://wiki.freebsd.org/NetworkRSS, 2014. Accessed: Jan 2017.

[26] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 55–73, 2024. https://www.usenix.org/conference/nsdi24/presentation/fried.

[27] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 281–297, 2020. https://www.usenix.org/conference/osdi20/presentation/fried.

[28] Fritz Kruger. CPU bandwidth - the worrisome 2020 trend. https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/, 2020. Accessed: 2021-06-09.

[29] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. Parking packet payload with p4. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 274–281, 2020. https://doi.org/10.1145/3386367.3431295.

[30] Junfeng Guo. [patch 01/13] net/idpf/base: introduce base code. https://yhbt.net/lore/dpdk-dev/LV2PR11MB5997FEBB6B1EC4B900EF6CFFF7379@LV2PR11MB5997.namprd11.prod.outlook.com/T/, 2022. Intel IDPF. Accessed: 2022-11-28.

[31] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[32] Ofer Hayut, Noam Bloch, Michael Kagan, and Ariel Shachar. Efficient delivery of completion notifications. https://patents.google.com/patent/US8924605B2, 2012. Accessed: Nov 2024.

[33] Ofer Hayut, Noam Bloch, Michael Kagan, and Ariel Shachar. Reducing size of completion notifications. https://patents.google.com/patent/US8959265B2, 2012. Accessed: Nov 2024.

[34] Internet engineering task force (IETF). https://www.ietf.org/.

[35] InfiniBand Trade Association (IBTA). What is InfiniBand. https://www.infinibandta.org/ibta-specification/. Accessed: Dec 2021.

[36] Intel. Processor Counter Monitor (PCM). https://github.com/opcm/pcm. Accessed: 2021-02-05.

[37] Intel. Tuning the buffers: a practical guide to reduce or avoid packet loss in dpdk applications. https://indeni.com/blog/cross-vendor-alert-of-the-week-some-received-packets-have-been-dropped-by-nic/, 2017. Accessed: Jun 2021.

[38] Intel Corporation. DPDK: Data plane development kit. http://dpdk.org, 2010. Accessed: May 2016.

[39] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 567–584, 2022. https://www.usenix.org/conference/nsdi22/presentation/iyer.

[40] Van Jacobson. Evolving from AFAP: Teaching NICs about time. https://legacy.netdevconf.info/0x12/session.html?evolving-from-afap-teaching-nics-about-time, 2018. Accessed: 2023-02-03.

[41] John McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/, 1996. Accessed: Mar 2024.

[42] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019. https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[43] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 171–186, 2018. https://www.usenix.org/conference/nsdi18/presentation/katsikas.

[44] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 67–81, 2016. http://dx.doi.org/10.1145/2872362.2872367.

[45] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019. https://www.usenix.org/conference/atc19/presentation/kogias-r2p2.

[46] Maciek Konstantynowicz, Patrick Lu, and Shrikant M. Shah. Benchmarking and analysis of software data planes. https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf, 2017. White paper from FD.io – The Fast Data I/O Project.

[47] Kevin Laatz. [dpdk-dev] [PATCH v2 0/3] Increase default RX/TX ring sizes. https://mails.dpdk.org/archives/dev/2018-January/086889.html, 2018. Intel DPDK. Accessed: Jun 2021.

[48] Xueming Li. [dpdk-dev] [patch v11 0/7] ethdev: introduce shared rx queue. https://lore.kernel.org/all/20211020075319.2397551-1-xuemingl@nvidia.com/, 2021. Accessed: 2023-04-13.

[49] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[50] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently offloading Intra-Server orchestration to NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1293–1308, 2023. https://www.usenix.org/conference/nsdi23/presentation/lin.

[51] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 270–282, 2020. https://doi.org/10.1145/3387514.3405868.

[52] Daniel Marcovitch et al. NVIDIA NIC architecture view on rxBisect, 2024. Private communication with NVIDIA NIC architects.

[53] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 175–186, 2014. http://doi.acm.org/10.1145/2619239.2626311.

[54] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. Disk|Crypt|Net: Rethinking the stack for high-performance video streaming. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 211–224, 2017. https://doi.org/10.1145/3098822.3098844.

[55] Marvell. FastLinQ 41000 Series Adapters. https://www.marvell.com/content/dam/marvell/en/public-collateral/ethernet-adaptersandcontrollers/marvell-ethernet-adapters-fastlinq-41000-series-user-guide.pdf, 2020. Accessed: Jun 2021.

[56] Rob McGuinness and George Porter. Evaluating the performance of software nics for 100-gb/s datacenter traffic control. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 74–88, 2018. https://doi.org/10.1145/3230718.3230728.

[57] Mellanox. Mellanox adapters programmer's reference manual (PRM). https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf, 2016. Accessed: Nov 2022.

[58] Mellanox. Connectx®-5 en card product brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf, 2017. Accessed: 2019-08-06.

[59] Microsoft. Introduction to receive side scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2017. Accessed: Jan 2020.

[60] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 247–258, 1996. https://doi.org/10.1145/232973.232999.

[61] NVIDIA. NVIDIA NICs performance report with DPDK 23.03. https://fast.dpdk.org/doc/perf/DPDK_23_03_NVIDIA_NIC_performance_report.pdf, 2023. Accessed: 2023-08-08.

[62] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019. https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[63] Amy Ousterhout, Jonathan Perry, Hari Balakrishnan, and Petr Lapukhov. Flexplane: An experimentation platform for resource management in datacenters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 438–451, 2017. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ousterhout.

[64] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC offloads. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–35, 2021. https://doi.org/10.1145/3445814.3446732.

[65] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The benefits of general purpose on-NIC memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1130–1147, 2022. https://doi.org/10.1145/3503222.3507711.

[66] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with shared receive rings. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 949–968, 2023. https://www.usenix.org/conference/osdi23/presentation/pismenny.

[67] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017. https://doi.org/10.1145/3132747.3132780.

[68] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to measure the killer microsecond. In *Proceedings of the Workshop on Kernel-Bypass Networks*, pages 37–42, 2017. https://doi.org/10.1145/3098583.3098590.

[69] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 200–215, 2023. https://doi.org/10.1145/3600006.3613137.

[70] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Ensō: A streaming interface for NIC-Application communication. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1005–1025, 2023. https://www.usenix.org/conference/osdi23/presentation/sadok.

[71] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A High-Speed stateful packet processing approach for tbps programmable switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1237–1255, 2023. https://www.usenix.org/conference/nsdi23/presentation/scazzariello.

[72] Ariel Shahar. NIC-CPU ring contention, 2022. Private communication with NVIDIA NIC architects.

[73] Shahaf Shuler. [dpdk-dev] [patch v2 2/2] net/mlx5: add rx and tx tuning parameters. https://mails.dpdk.org/archives/dev/2018-May/099834.html, 2018. Mellanox. Accessed: Nov 2022.

[74] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. Ioctopus: Outsmarting nonuniform dma. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 101–115, 2020. https://doi.org/10.1145/3373376.3378509.

[75] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient NIC packet scheduling. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 33–46. USENIX Association, 2019. https://www.usenix.org/conference/nsdi19/presentation/stephens.

[76] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *USENIX Annual Technical Conference (ATC)*, pages 347–353, 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi.

[77] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The nebula rpc-optimized architecture. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020. https://doi.org/10.1109/ISCA45697.2020.00027.

[78] Herbert Tom and de Bruijn Willem. Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2011. Accessed: 2020-03-05.

[79] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283–297, 2018. https://www.usenix.org/conference/nsdi18/presentation/tootoonchian.

[80] Tariq Toukan. [PATCH net-next 08/10] net/mlx4_en: Increase default TX ring size. https://www.mail-archive.com/netdev@vger.kernel.org/msg173779.html, 2017. Mellanox. Accessed: Jun 2021.

[81] VMware. Large packet loss in the guest os using vmxnet3 in esxi (2039495). https://kb.vmware.com/s/article/2039495, 2021. Accessed: Jun 2021.

[82] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 987–1004, 2023. https://www.usenix.org/conference/osdi23/presentation/wei-smartnic.

[83] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't forget the I/O when allocating your LLC. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021. https://doi.org/10.1109/ISCA52012.2021.00018.



XSched: Preemptive Scheduling for Diverse XPUs

Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen*, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

XPUs, such as GPUs, NPUs, ASICs, and FPGAs, lack flexible scheduling capabilities, failing to meet rich application requirements (e.g., priority and fairness) in multitasking environments. This paper presents XSched, a scheduling framework that enables preemptive scheduling on diverse XPUs with flexible policies. XSched provides unified interfaces for scheduling XPU tasks through a preemptible command queue abstraction (XQueue). The key challenge in implementing the abstraction is adapting to XPUs with diverse and evolving hardware capabilities and software stacks. XSched proposes a multi-level hardware model that enables mature, advanced XPUs to achieve optimal scheduling performance while maintaining compatibility with emerging, wimpy XPUs. To demonstrate the generalizability of XSched, we adapted it to ten XPUs of different types, brands, and generations across seven software platforms and implemented two hardware-agnostic scheduling policies. We further evaluated XSched through three case studies of multitasking workloads on XPUs. XSched effectively achieves various scheduling objectives using its efficient and flexible preemption mechanisms.

1 Introduction

"Providing education for all without discrimination;
Teaching students in accordance with their aptitudes."

— Confucius, *The Analects*

XPUs—referring to various accelerators, such as GPUs, NPUs, ASICs, and FPGAs—are extensively deployed in emerging systems, from cloud to edge, to offload intensive computations from host CPUs [52, 56, 64, 138]. These systems often involve multiple concurrent XPU tasks [26, 53, 102, 110, 121, 130]. Sharing XPUs among these tasks is a common practice to improve hardware efficiency and an inevitable choice for resource-constrained platforms. For example, cloud providers share a single GPU among multiple tenants to reduce costs [35, 131, 136], while edge platforms run multiple AI models on a single NPU [53, 68, 97, 105].

Rich application scenarios pose diverse requirements for XPU task scheduling. For instance, real-time systems (e.g., autonomous vehicles) need immediate responses for critical XPU tasks [2, 51, 134], while cloud providers prioritize fairness among tenants and maximum hardware utilization [126]. Although XPUs accelerate individual tasks effectively, they often struggle to meet application requirements under multitasking workloads due to inadequate scheduling support. The hardware scheduler baked into XPUs adopts either non-preemptive first-come, first-served (FCFS) scheduling (e.g., Intel NPUs [45], NVIDIA and AMD GPUs [75, 99]) or simple round-robin (RR) scheduling (e.g., multi-process GPU tasks [3]). This may lead to unfairness and priority inversions [60, 61, 100, 103, 113], which manifest as service-level objective (SLO) violations in data centers or missed deadlines in latency-critical autonomous systems. For example, in a video conferencing application (see §8), the tail latency of a real-time fake-background task increases by over 20× when running alongside a speech-to-text task on an Intel NPU [45].

Previous studies have proposed several software-based preemptive scheduling systems [16, 19, 39, 129] to bypass inadequate hardware schedulers, enabling the host CPU control over XPU task scheduling.[1] However, current solutions show a serious lack of **generalizability** when deployed across diverse XPUs of different types, brands, and generations, commonly found in modern cloud and edge computing platforms [26, 45, 53, 85, 126]. *Portability*: Existing preemptive scheduling systems [12, 19, 39, 129] are designed specifically for certain GPUs, making it challenging to port them to other accelerators like NPUs, ASICs, and FPGAs, or even to GPUs from different manufacturers and architectures [39]. To our knowledge, no software-based preemptive scheduling solutions exist for NPUs and ASICs. *Uniformity*: The lack of a unified abstraction for scheduling tasks on XPUs hinders the development of hardware-agnostic policies tailored to different application requirements. This also creates barriers to scheduling tasks on heterogeneous hardware with multiple XPUs, which is becoming increasingly popular in AI PCs [45] and autonomous devices [85]. *Evolvability*: The tight coupling between software and hardware implementations prevents them from evolving independently. This makes it difficult for existing scheduling systems to quickly integrate newly released or undocumented hardware features while flexibly excluding obsolete or disabled ones.

The root cause of the challenges in implementing general preemptive XPU scheduling is twofold. First, XPUs have significant and evolving differences in hardware capabilities. For example, while most systems rely on the general-purpose programmability of GPUs for preemptive scheduling through

*Corresponding author (rongchen@sjtu.edu.cn) and project leader

[1] This paper also considers preemption from the perspective of tasks, which typically comprises hundreds or thousands of XPU commands, e.g., kernels.

code transformation [12, 19, 39, 129], mainstream NPUs [82] and ASICs [89, 90] are non-programmable. Additionally, prior work [123] uses a specific `ioctl` to control timeslices of NVIDIA Tegra embedded GPUs [86], which is not available in desktop or server GPUs. Second, the software stacks of XPUs (e.g., user/kernel-mode drivers and firmware) are notoriously complex and highly customized. This often leads scheduling systems—both their mechanisms and policies—to become tightly coupled with specific details of the XPU. For example, TimeGraph [60] only works on Direct Rendering Infrastructure (DRI) drivers, and REEF [39] is integrated with the runtime and driver of AMD GPUs.

Key idea. We propose establishing a *unified abstraction* with a *multi-level hardware model* for preemptive XPU scheduling, which effectively conceals the differences and complexities in hardware capabilities and software stacks, ensuring generalizability. The abstraction provides unified interfaces for task scheduling across diverse XPUs, enabling hardware-agnostic policies and cooperation between XPUs. The model defines support levels aligning with fundamental requirements of preemptive scheduling, allowing developers to seamlessly implement one or more levels based on the hardware capabilities of each XPU and agilely evolve as capabilities change.

Our approach. We introduce XQueue, a *preemptible command queue* abstraction designed for XPU task scheduling. XQueue is analogous to CPU thread abstraction, as illustrated in Fig. 1. Each XQueue hosts an XPU task, which consists of a sequence of *commands*—such as GPU kernels, memory-copy operations, tensor operators, and other XPU-specific operations. An XPU functions as a worker, consuming commands from multiple XQueues. Task preemption is accomplished by switching between these XQueues. XQueue offers simple yet powerful interfaces for preemptive scheduling, making it easy to implement various hardware-agnostic policies, like fixed priority [74] and bandwidth partition [1].

We propose a *multi-level hardware model* to implement the XQueue abstraction on diverse XPUs. The model decouples preemption mechanisms from XPU-specific details and categorizes them into three levels. Level 1 (Lv1) allows XPUs to preempt (pending) commands that have been submitted but not launched. Level 2 (Lv2) extends Lv1 by enabling XPUs to preempt (in-flight) commands that have been launched but not executed. Level 3 (Lv3) builds upon Lv2 by allowing XPUs to preempt (running) commands that are being executed.

Based on our unified abstraction and multi-level hardware model, we implement XSched, a preemptive XPU scheduling framework. XSched is designed to be both efficient in scheduling XPU tasks and transparent to applications. For efficiency, XSched launches XPU commands *asynchronously* but *progressively* to pipeline command launch and execution for low task latency, while keeping most of the commands under host control for scheduling. For transparency, XSched provides a *shim* layer that intercepts commands from applications using

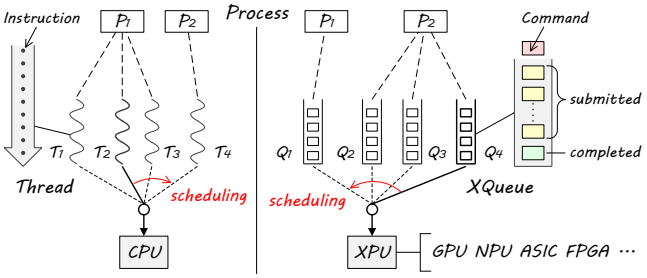


Fig. 1: *A unified abstraction for preemptive XPU scheduling.*

APIs of the original platform and forwards them to XSched. Moreover, we devise two new techniques for Lv2 preemptive scheduling on NVIDIA GPUs [128] and Intel NPUs [45], as well as one for Lv3 on NVIDIA GPUs released after the Pascal architecture [79].

To showcase the generalizability of XSched, we adapted it to different types, brands, and generations of XPUs across seven software platforms: NVIDIA (Kepler, Volta, and Ampere) [128], AMD [127], and Intel [45] GPUs; Ascend [43], Intel [45], and NVIDIA [82] NPUs; NVIDIA (PVA [90] and OFA [89]) ASICs; and Xilinx [4] FPGAs. Thanks to the multi-level hardware model, XSched makes it possible (even easy) to enable preemptive scheduling on emerging, wimpy XPUs, while fully leveraging the capabilities of mature, advanced XPUs. XSched is the first software-based system to support preemptive scheduling on these NPUs and ASICs. The Lv1 implementations require just 214–841 lines of C++ code and can be shared across XPUs supported by the same software platform. Additionally, we implemented Lv2 and Lv3 preemption on five and three categories of XPUs, respectively, using our new techniques with specific hardware capabilities.

We implemented two hardware-agnostic scheduling policies based on our unified abstraction and evaluated them on ten distinct XPUs against their native hardware schedulers. For the fixed priority policy [74], XSched reduces the tail latency (99^{th} percentile, P_{99}) of high-priority tasks by up to 2.10×. For the bandwidth partition policy [1], XSched partitions the XPU according to the throughput assigned for each task, with an acceptable overhead (1.5% on average). XSched also enables cooperative scheduling among XPUs, reducing the P_{99} latency of foreground NPU tasks by up to 2.63×, when running background tasks on both NPU and GPU.

We further evaluated XSched through three case studies. In a multi-tenant cloud scenario hosting two containers on a single GPU, XSched harvests 2.74× more GPU resources than TGS [130] while maintaining production container performance. In a video-conferencing application that runs two tasks on a single Intel NPU, XSched reduces the P_{99} frame latency by 9.26×. For multi-model inference serving, XSched reduces the P_{99} inference latency in Triton [94], a production-level system, by 30.0%, and achieves performance comparable to Paella [75], a state-of-the-art hardware-specific solution. Integrating XSched requires only a dozen lines of code.

2 Background and Motivation

2.1 Characterizing XPU Tasks

The XPU functions as a peripheral device that is managed by the host CPU. Despite the diversity of XPU architectures, runtimes, and programming models, XPU tasks share a common execution pattern. Typically, an XPU task consists of a sequence of *commands*, including GPU kernels, memory-copy operations, tensor operators on NPUs and TPUs, color space conversions on image-processing ASICs, and other XPU-specific operations. These commands execute sequentially based on data dependencies [22, 39, 75]. Additionally, auxiliary commands, such as CUDA and OpenCL events, are available to monitor command execution on XPUs. For each XPU task, the host CPU launches all its commands and then uses synchronization APIs to wait for their completion, since commands run asynchronously with it. Take a DNN inference task on a GPU as an example: the host CPU first initiates a memory-copy command to load the input into GPU memory. Then, it launches hundreds or thousands of GPU kernel commands to perform the inference layer by layer, followed by a memory-copy command to retrieve the results. These assumptions about XPUs and their tasks apply to most modern accelerators, including GPUs, NPUs, ASICs, and FPGAs.

2.2 Necessity of Preemptive Task Scheduling

XPUs are widely deployed across emerging systems—from cloud to edge—to offload intensive computations from host CPUs. Sharing XPU among multiple tasks enhances hardware utilization in cloud platforms and is essential for resource-constrained edge and mobile devices. For example, cloud providers commonly use a single GPU to serve multiple tenants or serverless function instances [26, 102, 130]. In autonomous vehicles, various algorithms, including perception, planning, and decision-making, are deployed on one TPU or ASIC [110]. On smartphones, both foreground and background AI tasks, like real-time voice input and photo indexing, run simultaneously on one NPU [53].

The demand for multi-task scheduling varies across different application scenarios. For instance, industrial automation applications like robotics and autonomous driving require low and deterministic latency [39, 41, 68]. Datacenters must ensure the service-level objective (SLO) [9, 36, 130, 139] and fairness among tenants [17]. Meanwhile, mobile devices like smartphones and AI PCs prioritize power efficiency [11, 31, 112] and user responsiveness [53]. Task preemption is a crucial mechanism to meet these diverse scheduling requirements [19, 22, 39, 57, 73, 101, 139], as it can significantly enhance system responsiveness and fairness while providing flexibility in scheduling.

2.3 Solution 1: XPU Hardware Scheduling

Lack of preemption support. Since XPUs typically function as peripherals, the host CPU launches commands sequentially to XPUs through a first-in, first-out (FIFO) ring buffer [39, 60]. This leads to tasks being naturally scheduled on a first-come, first-served (FCFS) basis. The non-preemptive policy is deeply embedded in the hardware, firmware, and drivers of various XPUs, including NVIDIA and AMD GPUs [39, 75, 109, 129], TPUs [22, 77, 106], Ascend and Intel NPUs [43, 45], and Jetson embedded ASICs [85]. Consequently, urgent tasks can be blocked by less critical ones, leading to priority inversions [28, 60, 129] and unpredictable task latencies. For example, we observed on different XPUs that sharing an XPU with just a background task can increase the tail latency of the foreground task by up to 2.19× (see §7.2). These issues are unacceptable in real-time scenarios like autonomous driving [2, 51, 134] and network packet processing [33, 59]. The non-preemptive policy also causes unfairness among multiple tenants in cloud environments [35, 136]. While some modern GPUs incorporate a simple time-sliced round-robin (RR) policy to serve tasks from different processes, latency-sensitive tasks still suffer from unpredictable performance degradation as the number of concurrent tasks increases [123].

Hard to support flexible policy. Hardware scheduling is restricted by its elementary, fixed policies. Due to real-time constraints and limited on-chip resources [133], native hardware schedulers cannot accommodate sophisticated policies. They also lack adequate runtime feedback and task information (e.g., priority and deadline), which are essential for implementing workload-aware scheduling strategies like those in SHEPHERD [139] and Paella [75]. Furthermore, since the policy is baked into the hardware or firmware, it cannot adapt agilely to changing workloads and scheduling requirements.

2.4 Solution 2: Host-managed XPU Scheduling

Scheduling XPU tasks on the host CPU is a preferred solution as it offers enhanced flexibility and customizability. Users can select from various scheduling policies tailored to application requirements, akin to traditional CPU thread scheduling.

Design for specific XPUs. Prior work proposed several host-managed scheduling approaches to bypass the inefficient XPU hardware scheduler and take over scheduling on the host CPU. However, these solutions lack generalizability due to their reliance on specific hardware capabilities. For example, EffiSha [19] and FLEP [129] enable preemption through GPU kernel transformation, which requires general-purpose programmability. A recent work [123] uses a specific `ioctl` interface to control timeslices of NVIDIA Tegra embedded GPUs. REEF [39] and a prior work [66] depend on a microcontroller function on AMD or ARM GPUs to reset compute units. Moreover, these solutions are tightly coupled with XPU-specific software stacks, further limiting their generalizability. For example, REEF [39] is integrated with the AMD GPU driver and compiler. TimeGraph [60] only works with drivers built on the Direct Rendering Infrastructure (DRI).

XPU-specific solutions bring issues with portability, uni-

formity, and evolvability, especially for users and developers of heterogeneous cloud and edge platforms that host different types, brands, and generations of XPUs. First, these solutions are difficult to port between different XPU types or across vendors and architectures. Second, they lack a unified abstraction for task scheduling across different XPUs, which impedes the development of hardware-agnostic policies and cooperative scheduling on heterogeneous platforms like Intel Core Ultra [45] and NVIDIA Jetson Orin [85]. Third, these solutions struggle to evolve with hardware changes, whether incorporating new features (e.g., interrupts on NVIDIA GPUs since Pascal architecture [75]) or handling deprecated ones.

Lack of unified abstraction and hardware model. The major challenge towards general solutions is the lack of unified abstraction and appropriate hardware models for diverse XPUs. A class of traditional hardware, like CPUs and disks, can be uniformly managed by the OS, regardless of their different architectures or manufacturers, as they share similar hardware capabilities that a general hardware model can adequately describe. For example, CPUs offer interrupts and privilege modes, while disks implement block device interfaces. Based on these models, the OS provides unified abstractions, like threads and files, to assist users in managing and utilizing the same class of devices.

For XPU scheduling, the hardware model conceals the complexity and heterogeneity of XPUs. Supporting new hardware requires only implementing the hardware model interfaces. Meanwhile, a unified abstraction for XPU tasks enables hardware-agnostic and cooperative scheduling policies by decoupling policies from task preemption mechanisms and providing a uniform view across different XPUs. The abstraction and hardware model keep the OS design clean and flexible to evolve. However, XPUs currently lack such unified abstraction and hardware model due to their significant and evolving differences in hardware capabilities.

3 The XQueue Abstraction

Opportunity. Despite the significant and evolving differences in their hardware capabilities and software stacks, XPU drivers in different software platforms generally provide queue-like programming model and interfaces, which we collectively refer to as *hardware queue* (hwQueue). Examples of these include GPU streams in CUDA [93] and HIP [5], ZE command queues in LevelZero [47], MTL command queues in Metal [6], ACL runtime streams in Ascend [43], CL command queues in OpenCL [62], and VPI streams for vision processing ASICs [91].

3.1 Preemptible Command Queue

We propose XQueue, a *preemptible command queue* abstraction for preemptive scheduling on diverse XPUs. XQueue is a queue of XPU commands that are executed sequentially in the order of submission. When an application process runs a task (e.g., DNN inference) on the XPU, it instantiates the task into a sequence of commands (e.g., GPU kernels, memory-copy operations, and tensor operators) and **submits** them to the XQueue in order. These commands execute asynchronously and sequentially on the XPU. The process can **wait** for the commands in an XQueue to complete. Note that a process can have multiple XQueues and submit different tasks to them.

Table 1: *Interfaces of the* XQueue *abstraction.*

XQueue **Interface**	**Description**
`submit(xq, cmd)`	Submit a command (*cmd*) to XQueue (*xq*)
`wait(xq, cmd)`	Wait for a given *cmd* in *xq* to complete
`suspend(xq)`	Suspend *xq* to pause task execution
`resume(xq)`	Resume *xq* to continue task execution

This abstraction is familiar to developers, since many XPUs have already adopted it. There is no semantic gap between XQueue and hwQueue in task execution, allowing XQueue to be seamlessly integrated with existing applications. XQueue differs from hwQueue in how it schedules submitted commands of a task. hwQueue is non-preemptible [19, 75, 129], meaning that once commands are submitted to the hwQueue, there is no practical way to pause them. In contrast, commands submitted to the XQueue can be preempted by the host CPU. Specifically, the host can **suspend** an XQueue to pause the running XPU task, yielding the XPU to other tasks. It can also **resume** the XQueue to continue task execution. This scheduling process is transparent to applications, which still perceive normal execution behavior of the XPU.

The XQueue abstraction is analogous to CPU thread abstraction, as illustrated in Fig. 1. An XPU task is a sequence of commands executed by an XQueue, while a CPU task is a sequence of instructions executed by a CPU thread. Both XPU tasks and CPU tasks are scheduled by suspending and resuming their corresponding XQueues and threads.[2] Moreover, multiple XQueues can run concurrently to achieve higher hardware utilization and overall throughput if the XPU supports spatial multiplexing mechanisms like GPU streams, which is akin to CPU hyper-threading. In summary, the XQueue abstraction preserves the programming semantics of the hwQueue, while providing familiar thread-like preemptive scheduling capabilities, which facilitates application compatibility and simplifies unified XPU task scheduling.

Interfaces. Table 1 lists the interfaces of XQueue. The XQueue provides a preemptible command queue abstraction for XPU task execution and preemptive scheduling. It offers `submit` and `wait` interfaces that enable applications to submit commands for asynchronous execution and wait for their completion on XPUs. Additionally, the XQueue provides `suspend` and `resume` interfaces, allowing the scheduler to control whether its commands can be executed on the XPU. These interfaces hide the implementation de-

[2]The hwQueue functions similarly to a kernel thread in CPU scheduling, where the actual task scheduling occurs (Fig. 1 omits this detail for brevity).

```
# fixed priority policy
void schedule(xq_status)
1   xqs = get_ready_xqueues(xq_status)
2   highest = find_highest_priority(xqs)
    # schedule ready XQueues with highest priority
3   for xq in xqs
4     if get_priority(xq) == highest then
5       resume(xq)  # resume highest-priority XQueue
6     else
7       suspend(xq)  # suspend others

# bandwidth partition policy
void schedule(xq_status)
8   current = get_running_xqueue(xq_status)
    # schedule next XQueue if timeslice expires
9   if timeslice_is_expired(current) then
10    next = get_next_ready_xqueue(xq_status)
11    suspend(current)  # suspend current XQueue
12    resume(next)  # resume next XQueue
13    timeslice = get_ratio(next) × QUANTUM
      # call schedule() when timeout
14    add_timer(timeslice)
```

Fig. 2: *Pseudocode of fixed-priority and bandwidth-partition scheduling policy implementations based on* XQueue *interfaces.*

tails of preemption mechanisms on diverse XPUs, enabling hardware-agnostic scheduling policies.

3.2 Scheduling Policy

XQueue interfaces are flexible for implementing various preemptive scheduling policies, such as fixed priority (FP) [74], shortest remaining time first (SRTF) [114], earliest deadline first (EDF) [74], and bandwidth partition (BP) [1]. Fig. 2 demonstrates the implementation of fixed priority and bandwidth partition scheduling using XQueue. In the fixed priority policy, each XQueue is assigned a priority that reflects the urgency of its task. The scheduling policy is triggered when XQueue status (`xq_status`) changes, such as when a new command arrives (making the XQueue *ready*), or when all commands complete (making the XQueue *idle*). The policy then finds and resumes ready XQueues with the highest priority while suspending others. This allows urgent tasks to preempt less critical ones. In the bandwidth partition policy, each XQueue is assigned a timeslice proportional to its allocated bandwidth. When a timeslice expires, the current XQueue is suspended, and the next one is resumed in a round-robin order. A timer is set to trigger rescheduling when the timeslice runs out. This policy ensures that tasks share the XPU utilization according to their specified ratios.

4 Multi-level Hardware Model

XQueue provides a unified abstraction for preemptive XPU scheduling. The hardware model plays a crucial role in implementing XQueue on different XPUs by decoupling the preemption mechanism from XPU-specific details. However, embracing both portability and high preemption performance presents a significant challenge. For instance, if the model

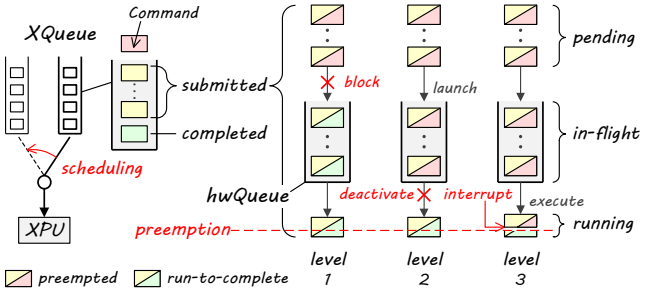


Fig. 3: *Comparison of three preemption levels for XPU scheduling.*

defines a preemption capability specific to certain XPUs to achieve optimal performance, it may not be portable to other XPUs that cannot support this capability.

Inspired by transaction isolation levels [13], we introduce a *multi-level hardware model* for preemptive scheduling on diverse XPUs. We characterize three levels of task preemption from the perspective of command states, as illustrated in Fig. 3. Each preemption level targets commands in one of three states: **pending** (ready to be launched[3]), **in-flight** (launched but not executed), or **running** (currently being executed). For each level, we identify the necessary hardware capabilities and interfaces to implement task preemption. Higher levels require more advanced capabilities for finer-grained preemption, while lower levels need only basic capabilities, making them applicable to a broader range of XPUs.

4.1 Level 1: Pending Command Preemption

Level 1 (Lv1) preemption targets commands in the pending state before launch. Once a command is launched, it is enqueued to the hardware queue (hwQueue), becoming in-flight and escaping host control. Therefore, the host can preempt pending commands by simply blocking their launch. Lv1 only requires capabilities to **launch** and **synchronize** commands, which all XPUs provide. The preemption latency is the total execution time of all launched commands, as shown in Fig. 3.

4.2 Level 2: In-flight Command Preemption

Unlike Lv1, which passively waits for in-flight commands to complete, Level 2 (Lv2) preemption can actively prevent in-flight commands from executing. This greatly reduces the preemption latency to just the execution time of the currently running command, as shown in Fig. 3. The key capability is to **deactivate** and **reactivate** the hwQueue. Once deactivated, no new commands from this hwQueue will execute until reactivation, enabling Lv2 preemption. This deactivation can be implemented by stalling-based or flushing-based approaches.

Stalling-based deactivation. The hwQueue can be deactivated by stalling command dequeuing, which prevents new commands from being fetched for execution. This approach requires advanced XPUs with integrated microcon-

[3]Launch refers to enqueuing a command to a hwQueue, after which launched commands escape host control—unlike the `submit` interface of XQueue.

Table 2: *Interfaces of the multi-level XPU hardware model.*

Level	Interface	Description
Lv1	`launch(hwq, cmd)`	Enqueue a given command (`cmd`) to a hwQueue (`hwq`) for executing it sequentially and asynchronously
	`sync(hwq, cmd)`	Wait for a given `cmd` in a `hwq` to complete
Lv2	`deactivate(hwq)`	Deactivate a given `hwq` to prevent all its commands from being selected for execution
	`reactivate(hwq)`	Reactivate a given `hwq` to allow all its commands to be selected for execution
Lv3	`interrupt(hwq)`	Interrupt the running command of a given `hwq`
	`restore(hwq)`	Restore the interrupted command of a given `hwq`

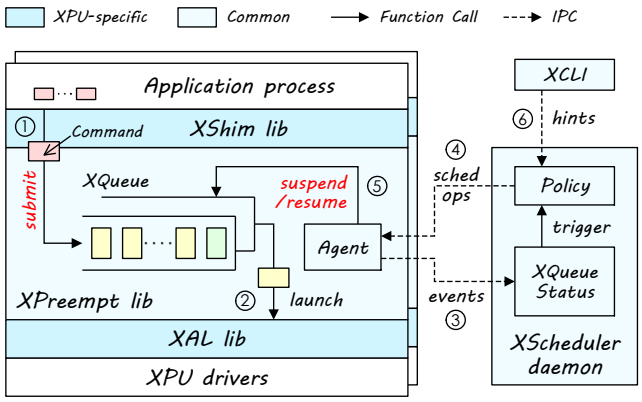


Fig. 4: *Architecture and workflow of* XSched.

trollers [30, 44, 46, 55, 72] that can selectively dequeue commands based on their attributes. The host can either instruct the microcontroller directly or modify command attributes to control dequeuing, achieving deactivation and reactivation.

Flushing-based deactivation. Another approach is to flush all in-flight commands in the hwQueue and relaunch them upon reactivation. Prior work [19, 39, 129] proposes that commands could be retrofitted to flush themselves, which solely requires the commands to be programmable. This approach works on all programmable XPUs, e.g., GPUs and many NPUs [43].

4.3 Level 3: Running Command Preemption

Lv2 preemption still requires waiting for the running command to complete, which leads to unpredictable preemption latency and may not meet strict real-time requirements for applications like automation [2, 51, 134] and networking [33, 59]. In contrast, Level 3 (Lv3) preemption targets the running command, aiming to achieve ultra-low and stable preemption latency, as illustrated in Fig. 3. This requires hardware capabilities that can **interrupt** and **restore** the running command. Once interrupted by the host, the running command is instantly paused and preempted for the execution of another command. The interrupted command is later restored to continue its execution. These capabilities are already present in modern GPUs [39, 44, 79].

4.4 Hardware Model Interfaces

The hardware model hides XPU hardware and driver differences through multi-level interfaces, as listed in Table 2. The three levels form a hierarchical design. The higher level introduces advanced preemption functionality and relies on the implementation of all lower levels to function properly. Lv1 interfaces consist of native hwQueue operations: `launch` for asynchronous command execution and `sync` for command synchronization. Lv2 interfaces include `deactivate` and `reactivate` to control the execution of in-flight commands in the hwQueue. Lv3 further introduces `interrupt` to instantly pause the running command and `restore` to resume the interrupted command.

5 The XSched Framework

5.1 Overview

We present XSched, a preemptive scheduling framework that implements XQueue based on the multi-level hardware model and supports diverse XPUs. As illustrated in Fig. 4, XSched consists of four key components: XPU shim (XShim), XPU task preemption (XPreempt), XPU adapter layer (XAL), and scheduler. XShim, XPreempt, and XAL are three dynamically linked libraries that are preloaded into the application process, while the scheduler runs as a shared system service daemon.

XShim. Typically, applications call driver APIs to launch XPU commands, with task scheduling handled entirely by the underlying hardware. The XShim library changes this workflow by intercepting XPU driver API calls and redirecting commands to the XQueue (① in Fig. 4). The approach provides transparency, allowing applications to run on XSched without modifications. The XShim library can be reused across XPUs that share the same driver. Note that the XShim library is optional when porting XSched to a new XPU, since applications can directly call XQueue interfaces instead.

XPreempt. The XPreempt library implements the interfaces of XQueue abstraction listed in Table 1, including `submit` and `wait` to support task execution, and `suspend` and `resume` to schedule XQueues. Commands submitted to the XQueue are buffered and launched to the XPU at a proper time (② in Fig. 4) to achieve task preemption. The XPreempt library contains an agent that watches the state of XQueue (e.g., ready or idle) and generates scheduling events to notify the scheduler (③ in Fig. 4) via inter-process communication (IPC). The agent is also responsible for applying the scheduling operations (e.g., suspend or resume an XQueue) received from the scheduler (⑤ in Fig. 4).

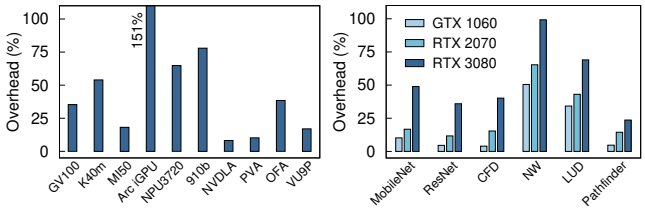


Fig. 5: *The performance overhead of synchronous execution (a) on various XPUs (GPUs/NPUs: ResNet-152 [40], OFA: Stereo Disparity [88], PVA: Gaussian Filter [88], VU9P: Vector Addition [132], and (b) using various deep learning (DL) [40, 42] and scientific computing [18] workloads on three NVIDIA GPUs.*

XAL. The XAL library implements the multi-level hardware model interfaces (listed in Table 2) by calling XPU driver APIs. Note that only Lv1 interfaces are mandatory for supporting a new XPU, while Lv2 and Lv3 interfaces are optional for better performance. Like XShim, the Lv1 implementation can be reused across XPUs sharing the same driver.

XScheduler. The scheduler runs as a daemon process, coordinating all XQueues from different processes. It monitors global XQueue status through agent-reported events (③ in Fig. 4), and invokes the scheduling policy to make decisions when status changes occur. The daemon performs these decisions by sending scheduling operations to agents (④ in Fig. 4). The policy in XScheduler is modular and customizable to suit various workloads. Users can change the policy and give scheduling hints (e.g., priority or bandwidth) through XCLI, a command-line tool (⑥ in Fig. 4).

5.2 XPreempt Design

The XPreempt library implements the preemptible XQueue abstraction using interfaces of our multi-level hardware model provided by the XAL, as shown in Fig. 6. Since the model conceals hardware details, it is challenging for XPreempt to enforce task preemption efficiently, particularly when using only Lv1 interfaces. This is because once commands are launched, they move beyond host control, and the scheduler can only passively wait for their completion.

Strawman: synchronous command execution. A straightforward solution [19, 61, 103, 111, 129] is to synchronize the XPU and block the host CPU after launching each command, preventing excess commands from being enqueued to the hwQueue. Consequently, the task is effectively preempted by simply blocking the host thread, and the preemption completes right after the currently running command finishes. However, this approach is highly inefficient for modern XPUs since it ignores their asynchronous nature. Synchronization is costly due to the communication latency between the CPU and the peripheral XPU, as well as driver overhead. More importantly, modern XPUs utilize asynchronous execution to pipeline command launching and execution, hiding the launching latency. Forcing synchronization creates frequent pipeline stalls (bubbles) that severely reduce command

```
# XQueue APIs
void submit(xq, cmd)
1   push_pending_cmd(xq, cmd)

void wait(xq, cmd)
2   if find_pending_cmd(xq, cmd) then
3     ...  # wait to launch cmd
4   sync(xq.hwq, cmd)  # sync for completion

void suspend(xq)
5   xq.mode = SUSPENDED  # suspend XQueue
6   if LEVEL >= 2 then deactivate(xq.hwq)
7   if LEVEL == 3 then interrupt(xq.hwq)

void resume(xq)
8   xq.mode = RUNNING  # resume XQueue
9   if LEVEL >= 2 then reactivate(xq.hwq)
10  if LEVEL == 3 then restore(xq.hwq)

# progressive command launching
void worker_thread(xq)
11  while TRUE
12    if get_inflight_cnt(xq) >= THRESHOLD then
        # wait until half of launched cmds complete
13      mid = get_inflight_middle(xq)
14      sync(xq.hwq, mid)  # sync for completion
15      pop_completed_cmds(xq)
16    cmd = pop_pending_cmd(xq)
17    while xq.mode == SUSPENDED
18      pause()  # pause cmd launching
19    launch(xq.hwq, cmd)  # launch cmd to hwqueue
20    push_inflight_cmd(cmd)
```

Fig. 6: *Pseudocode of* XQueue *APIs and progressive command launching implementation in* XPreempt.

throughput. As shown in Fig. 5 (a), synchronous command execution imposes substantial performance overhead on different XPUs, ranging from 8.2% to 151.3%. Furthermore, this overhead grows larger with more advanced XPUs, as pipeline stalls consume a greater proportion of time when XPU command execution speed increases. Fig. 5 (b) shows that the performance overhead increases by 3.04× on average when upgrading from GTX 1060 to RTX 3080.

Our solution: progressive command launching. Inspired by our prior work [39], XPreempt introduces a progressive command launching mechanism to balance preemption latency and runtime overhead, as shown in Fig. 6. This approach maintains a small number of in-flight commands, preventing pipeline stalls while enabling fine-grained preemption. When suspending the XQueue, the scheduler waits only for these few in-flight commands to complete, rather than all submitted commands (usually hundreds of commands).

Each XQueue contains a worker thread, a pending command buffer, an in-flight command log, and a corresponding hwQueue. When a command is submitted to the XQueue, it is first pushed into the pending command buffer (Line 1). The worker thread then *progressively* launches it to the hwQueue and records these in-flight commands in the log. Specifically, the worker checks the count of in-flight commands against a user-defined threshold (Line 12). If the count exceeds this

threshold, the worker invokes `sync` to wait for half of in-flight commands to complete (Lines 13–15). This threshold can be adjusted based on workloads to trade off preemption latency and runtime overhead. When the XQueue is suspended, the worker pauses to block new command launches (Lines 17–18). Once the XQueue is resumed, the worker continues to launch pending commands (Lines 19–20).

Lv2 and Lv3 preemption. For XPUs supporting Lv2 interfaces, XPreempt additionally deactivates the hwQueue after the worker is paused to accelerate task preemption when suspending the XQueue. Also, the hwQueue is reactivated when resuming. For XPUs supporting Lv3 interfaces, XPreempt further interrupts the currently running command in the hwQueue to preempt the task instantly when suspending the XQueue, and restores the command when resuming. The design of XPreempt is compatible with all three levels and can fully leverage the hardware capabilities thanks to the multi-level hardware model implemented in the XAL.

5.3 XScheduler Design

As depicted in Fig. 4, the XScheduler daemon is an event-driven service that coordinates all XQueues from different processes in the system, in cooperation with the XPreempt agents. Each agent monitors the change events of XQueue states, e.g., become *ready* when new commands are submitted and *idle* when all commands are completed. These events are sent to the XScheduler daemon to maintain global XQueue status (each XQueue: ready or idle, XPU device ID, process ID, etc.) and trigger the policy upon status changes. The policy decides which XQueues to suspend or resume based on current status. The decisions are sent back to the agents and applied by calling `suspend` and `resume` interfaces of these XQueues. Messages between XScheduler and the agents are passed via shared-memory IPC, enabling XSched to schedule XQueues across both processes and containers. For cross-VM scheduling, message passing can alternatively be implemented over network. This distributed XQueue design separates the control and data planes, minimizing command submission overhead and isolating errors within the application process.

The policy in XSched is designed to be flexible and customizable. XSched provides a `send_hint` API for applications and a command-line tool (XCLI) for users to provide hints to the policy, which set policy-specific parameters (e.g., priority, bandwidth, and deadline) of an XQueue. This is similar to how `setpriority` syscall and `nice` command set the priority in Linux. In addition to several built-in policies (e.g., fixed priority, bandwidth partition), users are free to customize policies optimized for their application scenarios. The policy should implement two basic interfaces: `schedule` and `recv_hint`, which are called when XQueue status changes and a new hint is given, respectively. As mentioned in Fig. 2, `schedule` uses `suspend` and `resume` to schedule XQueues, and `add_timer` to trigger itself after an interval.

6 Implementation on XPUs

We propose several new techniques to implement three preemption levels (XAL) on different XPUs, spanning various software platforms (CUDA, HIP, LevelZero, ACL, OpenCL, etc.), accelerator types (GPU, NPU, ASIC, and FPGA), and manufacturers (NVIDIA, AMD, Intel, Ascend, etc.).

6.1 Level 1 (Lv1) Preemption

Implementing Lv1 preemption is straightforward on XPUs since their drivers natively support hwQueue to launch and synchronize commands. Examples include `CUstream` in CUDA [93] (for NVIDIA GPUs), `hipStream` in HIP [5] (for AMD GPUs), `ze_command_queue` in LevelZero [47] (for Intel GPUs and NPUs), `aclrtStream` in ACL [43] (for Ascend NPUs), `VPIStream` in VPI [91] (for vision processing ASICs), and `cl_command_queue` in OpenCL [62] (for Xilinx FPGAs). The `launch` interface is implemented by calling the appropriate driver API corresponding to the command type, e.g., `cuLaunchKernel` for launching kernels on a `CUstream` and `cuMemcpyAsync` for memory copy commands. These drivers also support events, e.g., `CUevent` in CUDA and `cl_event` in OpenCL, which are fine-grained synchronization points that can be recorded on the hwQueue. The `sync` interface is implemented by synchronizing with an extra event recorded after a given command. If the driver does not support events, it can alternatively be implemented by synchronizing with the hwQueue. Since Lv1 implementation relies only on basic driver APIs, it can be shared across XPUs that use the same software platform. For example, the OpenCL implementation supports GPUs, FPGAs and even CPUs from various manufacturers.

6.2 Level 2 (Lv2) Preemption

XSched implemented hwQueue deactivation and reactivation using a hardware-assisted stalling approach on Intel NPUs and a software-based flushing approach on NVIDIA GPUs.

Stalling-based preemption. On-chip microcontrollers have been widely integrated into XPUs to selectively dispatch commands to execution units. Examples include Falcon microcontrollers in NVIDIA GPUs [10, 30, 55, 108], Command Processors in AMD GPUs [39], Graphics microcontrollers (GuCs) in Intel GPUs [44], LeonRT cores in Intel NPUs [46], and Taishan cores in Ascend NPUs [72]. These XPUs can preempt in-flight commands by instructing their microcontrollers to stall command dequeuing for deactivation. Recently, Intel released a new NPU firmware that supports these capabilities [48]. We modified the driver [116] to expose them to the host and implemented the Lv2 interfaces on Intel NPUs [45].

Flushing-based preemption. For XPUs without microcontroller support or exposed interfaces, we implement a software approach for flushing-based deactivation on programmable XPUs, demonstrating this on NVIDIA GPUs. XSched leverages command programmability, instead of hardware micro-

```
# GPU code injected at the beginning of kernel binary
void guardian(flag_ptr, hwq_id, cmd_id)
1     if *flag_ptr == DEACTIVATED then
2       push_aborted_cmd(hwq_id, cmd_id)
3       exit_thread()  # abort kernel execution

# host CPU code in XAL to implement Level 2 interfaces
void deactivate(hwq)
4     int *flag_ptr = get_deactivation_flag(hwq)
5     *flag_ptr = DEACTIVATED  # set per-hwqueue flag
void reactivate(hwq)
6     int *flag_ptr = get_deactivation_flag(hwq)
7     *flag_ptr = NORMAL # clear per-hwqueue flag
      # relaunch aborted cmds
8     while cmd = pop_aborted_cmd(hwq)
9       launch(hwq, cmd)
```

Fig. 7: *Pseudocode of flushing-based preemption implementation.*

controllers, to flush all in-flight commands in the hwQueue. As shown in Fig. 7, XSched prepends a *guardian* code snippet to each GPU kernel (i.e., command) that checks a per-hwQueue deactivation flag in GPU global memory. When this flag is set, the guardian code records the command ID and exits this command immediately. To deactivate the hwQueue, the host sets the flag, prompting all in-flight kernels to abort themselves. When reactivating, the host clears the flag and relaunches the aborted kernels.

XSched injects the guardian code snippet into each GPU kernel at runtime using dynamic binary instrumentation (DBI) technique [118]. This guardian code is compiled to binary using the NVIDIA compiler (NVCC) [83] and loaded into GPU instruction memory at process startup. As shown in Fig. 8, XSched rewrites the first instruction of each kernel with a `JMP` instruction to a per-kernel helper snippet. This snippet first loads the arguments of the guardian code from GPU constant memory, where kernel arguments are stored, and then calls the guardian code. After that, the snippet executes the original replaced instruction and returns to the next instruction in the kernel. Note that XSched leverages the hidden functions in the CUDA export table to allocate and access GPU instruction and constant memory.

Prior work proposed similar kernel-flushing schemes [19, 39, 129]. However, these approaches require modifications to kernel source code or PTX assembly, making them incompatible with closed-source frameworks (e.g., TensorRT [87]) or just-in-time compiled kernels (e.g., TensorFlow XLA [98]). To our knowledge, XSched introduces the first flushing-based deactivation at the binary level, allowing it to work seamlessly with all CUDA applications, including those built with closed-source CUDA kernel libraries (e.g., cuBLAS [81], cuDNN [84]) and inference frameworks like TensorRT.

6.3 Level 3 (Lv3) Preemption

Advanced modern XPUs, including NVIDIA GPUs released after the Pascal architecture [79], as well as recent AMD, Intel, and ARM GPUs [39, 44, 66], can interrupt running

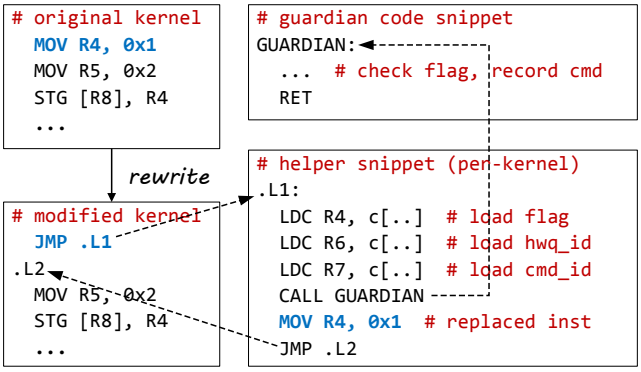


Fig. 8: *Binary instrumentation for flushing-based preemption.*

commands. We implemented Lv3 preemption on NVIDIA GPUs using two distinct approaches as illustrative examples.

TSG-based preemption. The interrupt mechanism on GPUs is designed for scheduling processes, or timeslice groups (TSGs) for NVIDIA GPUs [80, 123]. Each process is assigned a CUDA context corresponding to a TSG. When a TSG's timeslice expires, the GPU interrupts all running kernels in this TSG and switches to the next TSG in a round-robin manner. A previous study [123] found that TSGs of NVIDIA Tegra embedded GPUs could be adjusted through driver `ioctl` for task preemption. We implemented a similar approach on desktop and server GPUs (e.g., GV100), which dynamically adjusts TSG timeslices for Lv3 preemption. Specifically, the `interrupt` interface sets the timeslice to zero for the TSG containing the hwQueue to be preempted, and the `restore` interface resets the timeslice to its original value. Since this interrupt affects the entire TSG, this approach is only capable of inter-process scheduling. Note that the TSG-based approach achieves both Lv2 and Lv3 preemption by preempting both in-flight and running commands.

Queue-based preemption. We devise the first fine-grained interrupts on NVIDIA GPUs to implement Lv3 preemption at the hwQueue granularity. Although GPU interrupts are undocumented [75, 111], by tracing CUDA application syscalls, we discovered a specific `ioctl` that triggers GPU interrupts by writing to a GPU control register—a feature originally intended for kernel debugging. When the GPU is interrupted, it immediately stalls all running kernels and invokes the trap handler to save context and check trap reason. Using the DBI-based guardian technique for flushing-based preemption, XSched extends the trap handler to detect interrupts triggered by XSched. When detected, the target kernel is aborted for preemption, and the rest ones restore the context and continue execution. Unfortunately, NVIDIA GPUs do not expose support for resuming an aborted kernel from its interruption point. Inspired by prior work [39], XSched only preempts idempotent kernels and restarts them from the beginning.[4]

[4]We currently identify idempotent kernels manually, similar to prior work [39, 66, 70, 101]. A recent work [38] enables validation of idempotent kernels at launch time, and we plan to incorporate it to reduce manual efforts.

7 Experimental Evaluation

7.1 Portability

The support for preemptive scheduling on diverse XPUs. To demonstrate the portability of XSched, we adapted it to seven XPU platforms, as listed in Table 3. Thanks to our multi-level hardware model, only Lv1 implementation is necessary for basic preemptive scheduling, requiring just 214–841 lines of C++ code. Moreover, implementing Lv1 for a new XPU is straightforward, as most of the code simply unwraps and launches commands through the hwQueue provided by the drivers, sharing similar logic with just different function names. For example, implementing a memory copy command for the `cuMemcpyAsync` API in CUDA requires only 12 lines of code, and over 70% of the APIs follow a similar pattern. Additionally, the Lv1 implementation can be shared across XPUs that use the same software platform. For example, the implementation for OpenCL works seamlessly with FPGAs and GPUs from various manufacturers, including Xilinx, Intel, and Qualcomm [120]. For Lv2, we implemented stalling-based preemption on Intel NPUs and flushing-based preemption on NVIDIA Kepler, Volta, and Ampere GPUs. For Lv3, we implemented TSG-based preemption on NVIDIA GPUs released after the Pascal architecture and queue-based preemption on NVIDIA Volta and Ampere GPUs.

For performance experiments, we selected ten distinct XPUs: four GPUs (NV GV100 and K40m; AMD MI50; and Intel Arc iGPU [45]), three NPUs (Intel NPU3720 [45]; Ascend 910b [43]; and NV DLA [82]), two ASICs (NV PVA [90] and OFA [89]), and one FPGA (Xilinx VU9P [4]).

7.2 Uniformity

The support for flexible scheduling policy. Thanks to the unified abstraction of XSched, we implemented two hardware-agnostic and portable scheduling policies: fixed priority [74] and bandwidth partition [1], which required only 104 and 200 lines of C++ code, respectively. We evaluated them across ten distinct XPUs at their highest preemption level (see Table 3).

Workloads. In our experiments, we run two independent processes that submit the same types of tasks to a single XPU, with no shared data between them. We designate one process as foreground and the other as background, which the scheduling policy treats differently. For GPUs and NPUs, both processes submit DL inference tasks of the ResNet-152 model [40]. For ASICs, they execute the Stereo Disparity algorithm [88] on OFA, and the Gaussian Filter algorithm [88] on PVA. For FPGA, they perform vector addition tasks [132]. Note that commands are variably-sized, e.g., ranging from 2 to 113 μs for GV100, and 95 to 661 μs for DLA.

Fixed priority policy. The foreground process issues tasks periodically at a fixed frequency (20% of its peak throughput), and the background process issues tasks continuously at maximum frequency. XSched assigns a high priority to the foreground process and a low priority to the background process.

Table 3: *Description of XPUs and development effort (in lines of C++ code) for porting* XSched. ● *indicates a level that can be supported but has not yet been implemented, while* ○ *indicates a level that cannot be supported based on current hardware capabilities. Note that flushing-based Lv2 implementation on NVIDIA GPUs shares 513 LoCs for DBI, and TSG-based inter-process preemption takes 90 LoCs to implement Lv3 along with Lv2.*

Platform	XPU	XShim	Lv1	Lv2	Lv3
CUDA	NV Kepler GPUs	318	**511**	99 (+513)	○
	NV Volta GPUs			175 (+513)	301
	NV Ampere GPUs			189 (+513)	308
	NV GPUs w/ TSG			/	90
HIP	AMD GPUs	316	**841**	●	●
LevelZero	Intel GPUs	343	**379**	●	●
	Intel NPUs			131	○
ACL	Ascend NPUs	121	**260**	●	○
CUDLA	NV DLA	96	**247**	○	○
VPI	NV OFA, NV PVA	84	**214**	○	○
OpenCL	Xilinx FPGAs, GPUs and CPUs	204	**350**	●	○

Fig. 9 (top) shows the latency CDF of foreground tasks. The native hardware scheduler of all XPUs treats both processes equally, which doubles the tail latencies (99^{th} percentile, P_{99}) of foreground tasks compared to standalone execution (1.60× to 2.19×). In contrast, XSched preempts background tasks upon foreground task arrival. As a result, the P_{99} latencies of foreground tasks are close to standalone performance (1.02× to 1.30×) and are up to 2.11× lower than those experienced with the native hardware scheduler.

Bandwidth partition policy. Both the foreground and background processes continuously issue tasks at their maximum frequency. XSched allocates 75% of the XPU utilization to the foreground process and 25% to the background process. This simulates a scenario where an XPU is temporarily partitioned between two users with different QoS requirements [35]. Fig. 9 (bottom) shows the throughput of both tasks, normalized by the throughput of standalone execution. For all XPUs, the native hardware scheduler shares the XPU equally between foreground and background tasks. XSched achieves similar overall throughput to standalone execution with only 1.5% overhead on average, while guaranteeing the desired throughput partition ratio between two processes. On MI50, 910b, and VU9P, the native hardware scheduler surpasses both standalone execution and XSched in total throughput, as a single process cannot fully utilize hardware resources.

The uniform policy for heterogeneous XPUs. The XQueue interface facilitates seamless implementation of policies that cooperatively schedule multiple XPUs.

Workloads. The experiments are conducted on an NVIDIA Jetson Orin and an AI PC powered by Intel Core Ultra 9 185H. Initially, we run the same workload (i.e., two pro-

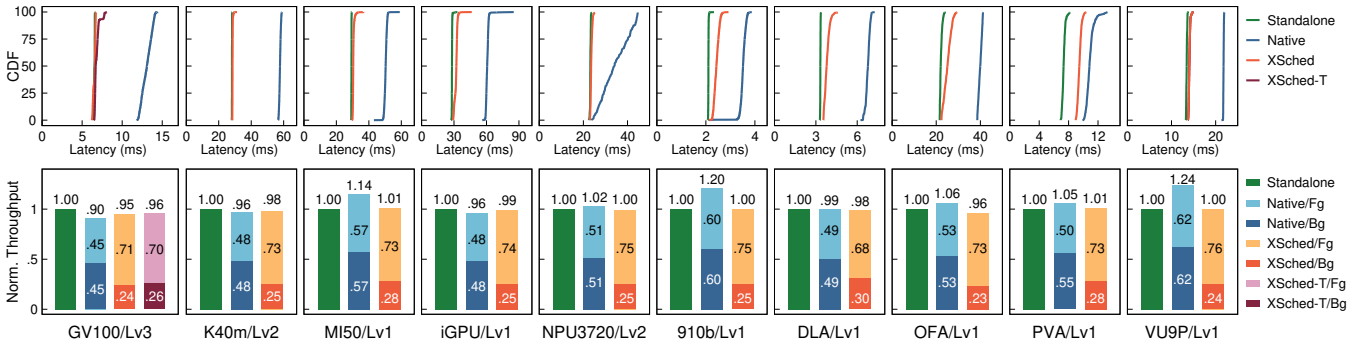


Fig. 9: *The scheduling performance of* XSched *on different XPUs. (Top): The latency CDF of the foreground task using fixed priority policy, (Bottom): The normalized throughput of foreground (Fg) and background (Bg) tasks using bandwidth partition policy.* XSched-T *refers to TSG-based inter-process preemption (§6.3). The thresholds for in-flight commands are tuned for each device, ranging from 2 to 16.*

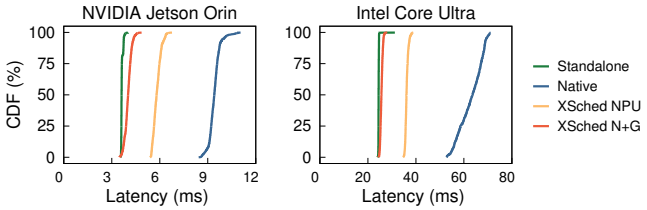


Fig. 10: *The latency CDF when co-running foreground NPU tasks with GPU and NPU background tasks on heterogeneous platforms.* 'XSched NPU' *means using* XSched *only for the NPU.* 'XSched N+G' *means using* XSched *for both NPU and GPU cooperatively.*

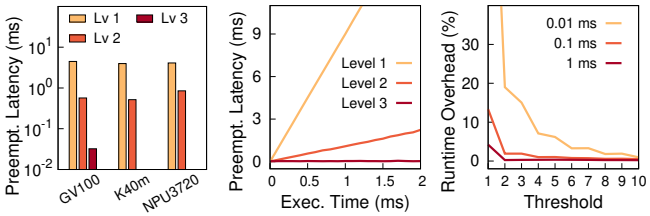


Fig. 11: *(a) The P_{99} preemption latency using different preemption levels, (b) the P_{99} preemption latency of GV100 for commands with different execution times, and (c) the runtime overhead of Lv1 on GV100 with different thresholds and command execution times.*

cesses running inference) as in Fig. 9 (top) for NPUs (i.e., NVIDIA DLA and Intel NPU3720). Subsequently, we run memory bandwidth-consuming tasks in another low-priority background process using the integrated GPU on the SoC.

Heterogeneous priority policy. Fig. 10 shows the CDF of the foreground task latencies. Although GPU and NPU operate independently for the computing resources, they may compete for the memory bandwidth and power supply on the SoC, leading to performance interference. Consequently, when scheduling only NPU tasks (XSched NPU), the latency, while reduced compared to the native hardware scheduler, remains significantly worse than the standalone case (1.67× and 1.55× for P_{99}). To address this, we implement a scheduling policy that uniformly schedules NPU and GPU tasks, allowing a high-priority XQueue to preempt other lower-priority ones, regardless of their associated XPU. With this policy (XSched N+G), the latency of foreground NPU tasks is close to standalone (1.18× and 1.09× for P_{99}), achieving a reduction of up to 2.63× compared to the native hardware scheduler.

7.3 Evolvability

The effect of advanced preemptive scheduling. The multi-level hardware model enables XPUs with advanced scheduling capabilities to further enhance the scheduling performance. As shown in Fig. 9 (top), XSched generally performs better on XPUs that support Lv2 and Lv3 interfaces (i.e., GV100, K40m, and NPU3720) compared to other XPUs that only support Lv1 interfaces. Specifically, the P_{99} latency of the foreground tasks on GV100, K40m and NPU3720 using XSched is only degraded by 1.9%, 3.8% and 5.4% compared to standalone execution. In contrast, the degradation is more significant on other XPUs, ranging from 7.3% to 29.6%.

We further evaluated the preemption latency of GV100, K40m, and NPU3720 with varying level configurations, as depicted in Fig. 11 (a). In this experiment, preempted tasks continually launch commands with an execution time of 0.5 ms (referred to as T hereafter), while the in-flight command threshold is set to 8. The P_{99} preemption latency with Lv1 support for all XPUs is approximately $8T$ due to the necessity to wait for all commands in the hwQueue to complete. This P_{99} latency is reduced to about $1T$ with Lv2, as only one command needs to be waited on. When employing Lv3 for GV100, the P_{99} preemption latency is further reduced to 32 μs, becoming independent of T. Fig. 11 (b) illustrates the P_{99} preemption latency of the GV100 with commands of varying execution durations, reinforcing this observed trend.

The adaptability to the evolution of XPUs. The multi-level hardware model also enables XSched to adapt to the evolving hardware capabilities and software interfaces of XPUs.

Hardware evolution. As many emerging XPUs are still under development, their hardware capabilities for preemption and scheduling may be enhanced in the subsequent generations. For example, the NVIDIA K40m does not support interrupt-based preemption [78, 79] for implementing Lv3 interfaces, while GV100 does. XSched can easily adapt to such

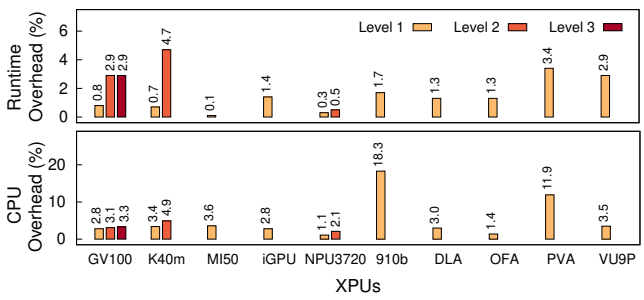


Fig. 12: *(a) The runtime overhead, and (b) the increased single-core CPU utilization of executing an XPU task with* XSched.

new hardware capability by *incrementally* implementing the higher levels of interfaces (e.g., Lv3 for GV100), while maintaining compatibility with the earlier generations of XPUs that only support lower levels of interfaces (e.g., Lv2 for K40m). We believe that for future generations of XPUs, XSched can continue to adapt to their evolved hardware capabilities.

Software evolution. The software interfaces of XPUs are also evolving rapidly. For example, XSched implements Lv2 for NPU3720 based on a new firmware released in July 2024 [48], seven months after the release of the hardware product. Additionally, the Lv3 implementation for GV100 relies on an undocumented and potentially unstable driver interface. If this interface is deprecated, XSched can selectively disable the Lv3 interfaces without altering other parts of the system.

7.4 Scheduling Overhead

Compared to the native hardware scheduler, XSched incurs additional overhead to support preemptive scheduling. We evaluate this overhead using the same tasks as in Fig. 9.

Runtime overhead. Fig. 12 (a) shows the runtime overhead of running an XPU task with XSched. The runtime overhead is less than 3.4% on all XPUs for Lv1, primarily due to the additional synchronizations required by the progressive command launching. For Lv2 on GV100 and K40m, the overhead increases by 2.1% and 4.0%, respectively, compared to Lv1. This is attributed to the execution of instrumented guardian code, whose performance is dominated by the latency of reading the flag. Therefore, XPUs with a high-performance memory (e.g., HBM2 on GV100) may show a less performance degradation compared to those with a low-performance memory (e.g., GDDR5 on K40m). In contrast, the hardware-assisted Lv2 on NPU3720 incurs no additional overhead.

Fig. 11 (c) further shows how in-flight command threshold and command execution time affect the runtime overhead of Lv1. Increasing the threshold can reduce runtime overhead as it reduces pipeline stalls. When the threshold exceeds 10, the overhead becomes negligible (less than 1%). The threshold in experiments is tuned to the minimum value that achieves overhead less than 3%. Moreover, shorter command execution time incurs more overhead due to the increased proportion of pipeline stalls, demonstrating the necessity of progressive launching as XPUs become faster.

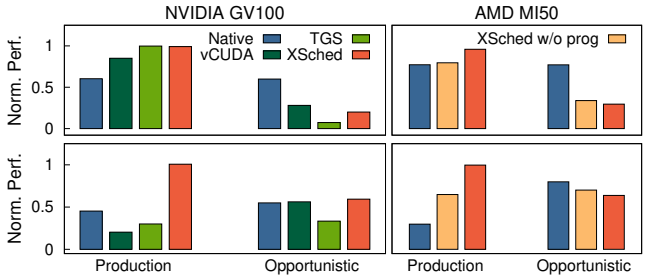


Fig. 13: *The scheduling performance of (Top) two containers running PyTorch training jobs, and (Bottom) financial algorithms as production jobs and scientific computing as opportunistic jobs. The performance is normalized by the standalone execution.*

CPU overhead. XSched employs several worker threads to manage queues and schedule XPUs, increasing CPU usage. As shown in Fig. 12 (b), XSched raises single-core CPU utilization by less than 5% in most cases, except for 910b (18.3%) and PVA (11.9%) because their drivers employ a spinning approach for synchronizing the hwQueue.

8 Case Studies

Next, we demonstrate the practical benefits of XSched through three case studies under different scenarios.

Case 1: GPU harvesting on multi-tenant server. To demonstrate the efficiency of XSched with its multi-level hardware model, we study how it manages two types of jobs on a single GPU: production jobs (Pjob) and opportunistic jobs (Ojob). Pjobs have stringent performance requirements with minimal degradation, while Ojobs should harvest remaining GPU resources on a best-effort basis.

We compare XSched against the native hardware scheduler, and two open-source GPU sharing systems, vCUDA [35, 115] and TGS [130]. Since vCUDA only supports quota-based configurations, we pre-profile the GPU utilization of Pjobs to allocate sufficient quota. TGS is evaluated without modifications, as it natively accommodates such workloads. XSched employs the fixed priority policy, assigning higher priority to Pjobs. The evaluation includes two workloads: (1) two containers running DL training jobs, the workload as in TGS, and (2) a container running financial algorithms (Black-Scholes [95]) as Pjob and a container running scientific computing (CFD [18]) as Ojob. We measure the request latency of financial algorithms and the throughput of other jobs.

Fig. 13 (left) shows the normalized performance of these systems on an NVIDIA GV100 GPU. Due to the lack of priority-based scheduling, vCUDA causes a 15.1% and a 80.0% performance degradation for Pjobs in DL training and financial algorithms, respectively. TGS can be regarded as a Lv1 implementation that carefully manages the number of in-flight commands for Ojobs by estimating the kernel submission rate of Pjobs. However, it presents two limitations. First, it misses the execution opportunities for Ojobs if the estimated kernel submission rate is inaccurate. Consequently,

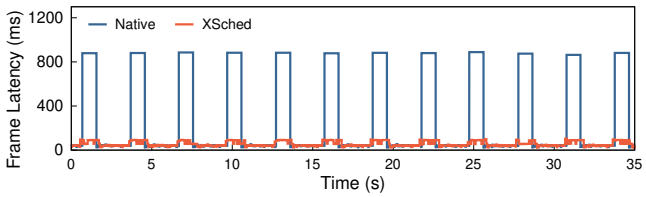


Fig. 14: *The frame latency of LFBW (fake-background) when running along with whisper.cpp (speech-to-text) on an Intel NPU3720.*

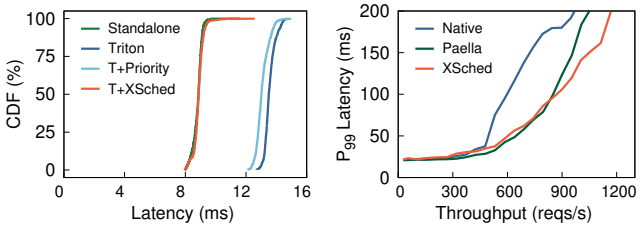


Fig. 15: *(a) The latency CDF of a high-priority Bert-large model inference using Triton and* XSched*, and (b) the throughput-latency curve of inference serving workloads using Paella and* XSched*.*

TGS can only harvest 7.3% of the GPU resources for Ojobs during DL training while XSched can utilize 20.0%, representing a 2.74× improvement. Second, TGS is effective only for Pjobs with specific patterns (e.g., DL training), and fails to accommodate other job types (e.g., financial algorithms). As a result, TGS exhibits a 70.0% performance degradation for Pjobs of financial algorithms. XSched overcomes these limitations by supporting Lv2 preemption, which mitigates the impact of in-flight commands from Ojobs on Pjobs' performance, thereby ensuring the performance of Pjobs (1.0% degradation) while improving the GPU utilization of Ojobs.

We further conducted the experiments on an AMD MI50 GPU, shown in Fig. 13 (right). Although XSched only implements Lv1 interfaces on AMD GPUs, it achieves a 4.1% and 0.4% performance degradation for Pjobs of DL training and financial algorithms, respectively. We also evaluated the performance of XSched without the progressive command launching technique (i.e., using synchronous command execution, denoted as XSched w/o prog), demonstrating its effectiveness for XPUs with only Lv1 implementations.

Case 2: Video conferencing on AI PC. In this case, we demonstrate the flexibility of XSched's policy by scheduling two applications simulating a video conferencing scenario: fake-background (LFBW [29]) and speech-to-text (whisper.cpp [32]). LFBW blurs the background of a video stream at a rate of 25 FPS. Whisper.cpp transcribes an audio stream every 3 seconds. Both applications run an AI model on Intel NPU3720 within an AI PC [45].

Fig. 14 shows the frame latency (i.e., the time between consecutive frames) of LFBW when running alongside whisper.cpp. Without XSched, the frame latency of LFBW is unstable and the P_{99} frame latency spikes up to 880 ms (20.12× higher than standalone), resulting in frequent frame drops and a jittery experience. This instability is due to the native hardware scheduler of the NPU3720 adopting a FCFS policy, causing LFBW to wait for the completion of whisper.cpp (0.8 s). To address this issue, we first attempted to use the fixed priority policy of XSched to prioritize LFBW. However, although the frame rate stabilized at 25 FPS, the latency of whisper.cpp significantly increased and exceeded its period, leading to the loss of transcribed text content. Therefore, we further implemented a laxity-based policy [96], which aims to maximize LFBW's frame rate while ensuring that whisper.cpp can complete within 3 s. Ultimately, the P_{99} frame latency of LFBW is reduced to 95 ms, an improvement of 9.26× compared to the native hardware scheduler, while ensuring that whisper.cpp did not lose any content.

Case 3: Multi-model inference serving. XSched is easy to be integrated into existing systems, thanks to the clean and unified XQueue interface. We demonstrate this by integrating XSched into two GPU-based inference serving systems to achieve low-latency inference through preemptive scheduling. The experiments are conducted on a single GV100.

Triton. We integrated XSched into a production-level inference serving system, Triton [94], with only 10 lines of code changes, to enable scheduling using the fixed priority policy of XSched. Specifically, we modified Triton to submit the scheduling hints to XSched when enqueueing inference requests. Triton allows users to specify the priority of a model. XSched retrofits this setting to assign the priority of the XQueues. To illustrate the effectiveness of integrating XSched, we used two clients to send inference requests for two Bert-large models [25]. The high-priority client sends requests with a frequency of 10 reqs/sec, while the low-priority client sends requests continuously. Fig. 15 (a) shows the latency CDF of the high-priority model. The vanilla Triton, which employs the native hardware scheduler, and Triton with priority settings (T+Priority) exhibit a 1.53× and 1.51× higher P_{99} latency compared to standalone execution. By integrating XSched (T+XSched), the P_{99} latency of the high-priority model is only 1.07× higher than standalone, a reduction of 30.0% compared to vanilla Triton.

Paella. We next compare the performance of XSched with Paella [75], a state-of-the-art inference serving system with GPU-specific scheduling techniques. We integrated XSched into the naive baseline (referred to as CUDA-MS in Paella's paper [75]) from the Paella artifact [76], which employs native hardware scheduling. The integration requires only 15 lines of code changes. We implemented a K-earliest deadline first (K-EDF) policy, which executes the K tasks with the earliest deadlines in parallel. K is set to 16 in this experiment. Fig. 15 (b) shows the throughput-latency curve. The workload used is consistent with that described in Paella's paper, where each client sends requests based on a log-normal distribution with a standard deviation of $\sigma = 2.0$. XSched achieves similar P_{99} latency as Paella at a lower throughput, and even outperforms Paella by 1.3× at a high throughput (1,000 reqs/sec).

9 Discussion and Limitations

Command-based offloading. Most XPUs function as peripheral devices passively managed by the host CPU, which issues commands to offload XPU tasks. Some XPUs, such as DPUs [14] and certain FPGAs [49, 117], can proactively execute tasks without host intervention. For these XPUs, host-managed scheduling solutions [19, 39, 123, 129], including XSched, are not applicable. One potential solution is to integrate XSched into the control unit of these XPUs, such as the ARM cores on DPUs [14].

Single-command tasks. Lv1 and Lv2 preemption mechanisms are designed for tasks that contains multiple commands, such as GPU kernels, memory-copy operations, and tensor operators. For tasks containing only a single command, such as CUDA graph or model inference on certain NPUs, developers need to either implement Lv3 or divide the task into multiple fine-grained commands through techniques like model slicing [37, 105] to enable fast task preemption.

Sufficient XPU memory. XSched currently focuses solely on computation scheduling and assumes sufficient XPU physical memory to hold all task data. On-demand paging mechanisms like CUDA Unified Memory [92], DeepUM [58], and SUV [7] can handle memory oversubscription. XSched could work together with these systems to support memory swapping with task scheduling, and we leave it to future work.

Untrusted tenants. XSched relies on applications to submit commands via XQueue APIs or use XShim to intercept those commands. However, a malicious tenant could bypass XSched or submit excessive commands to monopolize the XPU. Fortunately, XPU virtualization based on API remoting [27, 107, 122] can prevent tenants from direct XPU access. By integrating XSched into the hypervisor's API remoting server, all commands to the XPU can be properly managed.

10 Related Work

XPU hardware scheduling. Prior work proposed hardware-based scheduling techniques for various XPUs, including GPUs [73, 101, 124, 125, 135], TPUs [8, 137], NPUs [22, 24, 63, 67, 133], and FPGAs [57, 65, 69, 104]. Some are aimed to enhance the scheduling mechanisms of XPUs, e.g., fast kernel preemption [22, 73, 101] or efficient spatial sharing [124, 133]. XSched is orthogonal to these techniques, and the multi-level hardware model allows XSched to integrate these hardware enhancements. Some other work enhances the scheduling policy of the hardware scheduler, e.g., deadline-aware policy [135] or QoS-based policy [125]. These techniques require hardware modifications, while XSched can implement flexible software-defined policies.

Host-managed XPU scheduling. Prior work proposed various host-managed XPU scheduling techniques. Some aim to improve hardware utilization through GPU spatial multitasking [20, 21, 23, 50, 71, 119], while others focus on GPU task preemption [15, 19, 39, 54, 66, 129, 140]. For example, PTask [103], TimeGraph [60], Gdev [61], vCUDA [35, 115], and TGS [130] schedule pending GPU commands (Lv1) by restricting the command launching rate. EffiSha [19] and FLEP [129] preempt unexecuted GPU kernel parts (Lv2) by transforming the kernel source code to voluntarily abort inactive GPU thread blocks. REEF [39] supports running kernel preemption (Lv3) by modifying the AMD GPU driver to reset the whole task, yet is incompatible with closed-source systems and non-GPU hardware. Unlike prior work that focused on specific hardware platforms, XSched is the first general scheduling framework to support commodity GPUs, NPUs, ASICs, and FPGAs. Moreover, XSched can seamlessly incorporate new task preemption techniques (see §6) as implementations for their appropriate preemption levels.

Unified programming models and interface. Previous efforts like OpenCL [62], SYCL [34], and LevelZero [47] (oneAPI) also aimed to provide unified programming models and interfaces across different XPUs. However, hardware differences have led to numerous vendor-specific extensions, fragmenting their ecosystems. These platforms face a fundamental limitation—by defining only basic interfaces to maintain compatibility with common hardware capabilities of different XPUs, they restrict themselves to minimal functionality. In contrast, XSched introduces a multi-level hardware model to harness different capabilities of diverse XPUs while providing a unified abstraction.

11 Conclusion

Modern hardware accelerators (XPUs) are experiencing unprecedented growth, with numerous types, brands, and generations deployed across cloud and edge computing platforms. Their diverse hardware capabilities and complex software stacks expose unique challenges in designing and implementing preemptive scheduling with excellent generalizability. To tackle this, XSched proposes a unified XQueue abstraction with three-level hardware models, enabling mature, advanced XPUs to achieve optimal performance while maintaining compatibility with emerging, wimpy XPUs. We have adapted XSched to various XPU platforms and deployed it on a dozen XPUs to support preemptive scheduling with flexible policies. Our experimental evaluation and case studies demonstrate the efficacy and efficiency of XSched and its novel techniques. XSched, with its diverse XPU implementations, is open-source and publicly available at https://github.com/XpuOS/xsched.

Acknowledgments

We sincerely thank our anonymous shepherd and reviewers for their insightful comments and suggestions. This work was supported in part by the National Natural Science Foundation of China (No. 62432010, 62272291) and a research grant from Huawei Technologies.

References

[1] L. Abeni and G. Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. 4–13. https://doi.org/10.1109/REAL.1998.739726

[2] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. 2020. Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 267–280.

[3] AMD. 2024. Micro Engine Scheduler Specification. https://gpuopen.com/download/documentation/micro_engine_scheduler.pdf.

[4] AMD. 2025. AMD Virtex UltraScale+ FPGAs. https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/virtex-ultrascale-plus.html.

[5] AMD. 2025. HIP documentation. https://rocm.docs.amd.com/projects/HIP/en/latest/index.html.

[6] Apple. 2025. MTLCommandQueue. https://developer.apple.com/documentation/metal/mtlcommandqueue.

[7] Pratheek B, Guilherme Cox, Jan Vesely, and Arkaprava Basu. 2024. SUV: Static Analysis Guided Unified Virtual Memory. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture*. 293–308. https://doi.org/10.1109/MICRO61859.2024.00030

[8] Eun-Tae Baek, Dongup Kwon, and Jangwoo Kim. 2020. A Multi-Neural Network Acceleration Architecture. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*. 940–953.

[9] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 499–514. https://www.usenix.org/conference/osdi20/presentation/bai

[10] Joshua Bakita and James H. Anderson. 2023. Hardware Compute Partitioning on NVIDIA GPUs. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*. 54–66.

[11] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In *Proceedings of the 2016 IEEE/ACM International Symposium on Code Generation and Optimization*. 70–81.

[12] C. Basaran and K. Kang. 2012. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*. 287–296.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[14] Idan Burstein. 2021. Nvidia Data Center Processing Unit (DPU) Architecture. In *Proceedings of the 2021 IEEE Hot Chips Symposium*. 1–20.

[15] Jon C. Calhoun and Hai Jiang. 2012. Preemption of a CUDA Kernel Function. In *Proceedings of the 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. 247–252.

[16] N. Capodieci, R. Cavicchioli, M. Bertogna, and Aingara Paramakuru. 2018. Deadline-Based Scheduling for GPU with Preemption Support. In *Proceedings of the IEEE Real-Time Systems Symposium*. 119–130.

[17] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*. New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/3342195.3387555

[18] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 44–54.

[19] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[20] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[21] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*.

[22] Yujeong Choi and Minsoo Rhu. 2019. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*. 220–233.

[23] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[24] Anup Das. 2022. Real-Time Scheduling of Machine Learning Operations on Heterogeneous Neuromorphic SoC. In *Proceedings of the 20th ACM/IEEE International Conference on Formal Methods and Models for System Design*. 1–12.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:cs.CL/1810.04805 https://arxiv.org/abs/1810.04805

[26] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[27] José Duato, Antonio J. Peña, Federico Silla, Juan C. Fernández, Rafael Mayo, and Enrique S. Quintana-Ortí. 2011. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 18th International Conference on High Performance Computing*. 1–10. https://doi.org/10.1109/HiPC.2011.6152718

[28] Glenn A. Elliott and James H. Anderson. 2011. Real-World Constraints of GPUs in Real-Time Systems. In *Proceedings of the 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, Vol. 2. 48–54.

[29] Fufu Fang. 2025. Linux-Fake-Background-Webcam. https://github.com/fangfufu/Linux-Fake-Background-Webcam.

[30] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, and Shinpei Kato. 2013. Exploring microcontrollers in GPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*. New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/2500727.2500742

[31] Xiang Gao. 2023. TAS: A Temperature-Aware Scheduling for Heterogeneous Computing. *IEEE Access* 11 (2023), 54773–54781.

[32] ggml. 2025. whisper.cpp. https://github.com/ggml-org/whisper.cpp.

[33] Younghwan Go, Muhammad Asim Jamshed, Younggyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

[34] Khronos Group. 2025. SYCL: C++ Programming for Heterogeneous Parallel Computing. https://www.khronos.org/sycl/.

[35] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. 2018. GaiaGPU: Sharing GPUs in Container Clouds. In *Proceedings of the 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications*. 469–476.

[36] A. Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

[37] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. 2019. MOSAIC: Heterogeneity-, Communication-, and Constraint-Aware Model Slicing and Execution for Accurate and Efficient Inference. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 165–177. https://doi.org/10.1109/PACT.2019.00021

[38] Mingcong Han, Weihang Shen, Guanwen Peng, Rong Chen, and Haibo Chen. 2024. Microsecond-scale Dynamic Validation of Idempotency for GPU Kernels. arXiv:cs.OS/2410.23661 https://arxiv.org/abs/2410.23661

[39] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[40] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[41] Sabir Hossain and Deok Jin Lee. 2019. Deep Learning-Based Real-Time Multiple-Object Detection and Tracking from Aerial Imagery via a Flying Robot with GPU-Based Embedded Devices. *Sensors* 19 (2019).

[42] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:cs.CV/1704.04861 https://arxiv.org/abs/1704.04861

[43] Huawei. 2025. Huawei Atlas AI Platform. https://www.hiascend.com/en/hardware/product.

[44] Intel. 2024. Enabling the GuC/HuC Firmware for Linux on New Intel GPU Platforms. https://www.intel.com/content/www/us/en/content-details/609249/enabling-the-guc-huc-firmware-for-linux-on-new-intel-gpu-platforms.html.

[45] Intel. 2024. Intel Core Ultra Processors Family. https://www.intel.com/content/www/us/en/products/details/processors/core-ultra.html.

[46] Intel. 2024. Intel Core Ultra Processors (PS Series) Datasheet. https://www.intel.com/content/www/us/en/content-details/819636/intel-core-ultra-processors-ps-series-datasheet.html.

[47] Intel. 2024. Level Zero Specification documentation–Core Programming Guide. https://oneapi-src.github.io/level-zero-spec/level-zero/latest/core/PROG.html.

[48] Intel. 2024. Linux NPU Driver v1.5.1. https://github.com/intel/linux-npu-driver/releases/tag/v1.5.1.

[49] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a box: inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. USA, 425–438.

[50] Saksham Jain, Iljoo Baek, Shige Wang, and R. Rajkumar. 2019. Fractional GPUs: Software-Based Compute and Memory Bandwidth Reservation for GPUs. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 29–41.

[51] Won-Seok Jang, Hansaem Jeong, Kyungtae Kang, Nikil D. Dutt, and Jong-Chan Kim. 2020. R-TOD: Real-Time Object Detector with Minimized End-to-End Delay for Autonomous Driving. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*. 191–204.

[52] Jinwoo Jeon, Sungwook Jung, Eungchang Mason Lee, Duckyu Choi, and Hyun Myung. 2021. Run Your Visual-Inertial Odometry on NVIDIA Jetson: Benchmark Tests on a Micro Aerial Vehicle. *IEEE Robotics and Automation Letters* 6 (2021), 5332–5339.

[53] Joo Seong Jeong, Jingyu Lee, Dong-Hyun Kim, Chang-Kyu Jeon, Chang-Sung Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-DNN inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*.

[54] Zhuoran Ji and Cho-Li Wang. 2021. CTXBack: Enabling Low Latency GPU Context Switching via Context Flashback. In *Proceedings of the 35th IEEE International Parallel and Distributed Processing Symposium*. 121–130. https://doi.org/10.1109/IPDPS49936.2021.00021

[55] Joe Xie. 2016. NVIDIA RISC-V Story. https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf.

[56] Norman P. Jouppi, George Kurian, Sheng Li, Peter C. Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiaoping Zhou, Zongwei Zhou, and David A. Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

[57] Slavisa Jovanovic, Camel Tanougast, and Serge Weber. 2007. A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*. 358–364.

[58] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 207–221. https://doi.org/10.1145/3575693.3575736

[59] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. 2015. Raising the Bar for Using GPUs in Software Packet Processing. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

[60] Shinpei Kato, Karthik Lakshmanan, Ragunathan Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proceedings of the 2011 USENIX Annual Technical Conference*.

[61] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A. Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Annual Technical Conference*.

[62] Khronos OpenCL Working Group. 2025. The OpenCL Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

[63] Alexandros Kouris, Stylianos I. Venieris, Stefanos Laskaridis, and Nicholas Donald Lane. 2022. Fluid Batching: Exit-Aware Preemptive Serving of Early-Exit Neural Networks on Edge NPUs. arXiv:cs.LG/2209.13443 https://arxiv.org/abs/2209.13443

[64] Bertalan Kovács, Anders D. Henriksen, Jonathan Dyssel Stets, and Lazaros Nalpantidis. 2021. Object Detection on TPU Accelerated Embedded Devices. In *Proceedings of the 13th International Conference of Computer Vision Systems*, Markus Vincze, Timothy Patten, Henrik I. Christensen, Lazaros Nalpantidis, and Ming Liu (Eds.), Vol. 12899. 82–92. https://doi.org/10.1007/978-3-030-87156-7_7

[65] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. 2021. Compiler-driven FPGA virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[66] Hyeonsu Lee, Hyunjun Kim, Cheolgi Kim, Hwansoo Han, and Euiseong Seo. 2021. Idempotence-Based Preemptive GPU Kernel Scheduling for Embedded Systems. *IEEE Trans. Comput.* 70, 3 (2021), 332–346. https://doi.org/10.1109/TC.2020.2988251

[67] Jounghoo Lee, Jinwoo Choi, Jaeyeon Kim, Jinho Lee, and Youngsok Kim. 2021. Dataflow Mirroring: Architectural Support for Highly Efficient Fine-Grained Spatial Multitasking on Systolic-Array NPUs. In *Proceedings of the 58th ACM/IEEE Design Automation Conference*. 247–252.

[68] TIMOTHY B. LEE. 2019. Tesla's autonomy event: Impressive progress with an unrealistic timeline. https://arstechnica.com/cars/2019/04/teslas-autonomy-event-impressive-progress-with-an-unrealistic-timeline/.

[69] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. 2010. Hardware Context-Switch Methodology for Dynamically Partially Reconfigurable Systems. *J. Inf. Sci. Eng.* 26 (2010), 1289–1305.

[70] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran Monfort, Pradip Bose, Quan Chen, Minyi Guo, and Vijay Janapa Reddi. 2020. Asymmetric Resilience: Exploiting Task-Level Idempotency for Transient Error Recovery in Accelerator-Based Systems. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*. 44–57.

[71] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, R. Goh, and Deming Chen. 2015. Efficient GPU Spatial-Temporal Multitasking. *IEEE Transactions on Parallel and Distributed Systems* 26 (2015), 748–760.

[72] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *Proceedings of the 2019 IEEE Hot Chips Symposium*. 1–44.

[73] Zhen Lin, L. Nyland, and Huiyang Zhou. 2016. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 898–908.

[74] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

[75] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*.

[76] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2024. The Artifact for Paella. https://github.com/eniac/paella.

[77] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P. Jouppi, and David A. Patterson. 2020. Google's Training Chips Revealed: TPUv2 and TPUv3. In *Proceedings of the 2020 IEEE Hot Chips Symposium*. 1–70.

[78] NVIDIA. 2014. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210 Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[79] NVIDIA. 2016. Tuning CUDA Applications for Pascal: Compute Preemption. https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#compute-preemption

[80] NVIDIA. 2019. NVIDIA open-gpu-doc repository. https://github.com/NVIDIA/open-gpu-doc/blob/master/manuals/volta/gv100/dev_ram.ref.txt.

[81] NVIDIA. 2024. cuBLAS. https://developer.nvidia.com/cublas.

[82] NVIDIA. 2024. Deep Learning Accelerator (DLA). https://developer.nvidia.com/deep-learning-accelerator.

[83] NVIDIA. 2024. NVIDIA CUDA Compiler Driver NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[84] NVIDIA. 2024. NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[85] NVIDIA. 2024. NVIDIA Jetson Orin. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[86] NVIDIA. 2024. NVIDIA Jetson Xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/.

[87] NVIDIA. 2024. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[88] NVIDIA. 2024. Vision Programming Interface: Algorithms. https://docs.nvidia.com/vpi/algorithms.html.

[89] NVIDIA. 2024. VPI – Vision Programming Interface Backends: OFA. https://docs.nvidia.com/vpi/architecture.html#autotoc_md12

[90] NVIDIA. 2024. VPI – Vision Programming Interface Backends: PVA. https://docs.nvidia.com/vpi/architecture.html#autotoc_md10

[91] NVIDIA. 2024. VPI – Vision Programming Interface: Streams. https://docs.nvidia.com/vpi/architecture.html#arch_stream

[92] NVIDIA. 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[93] NVIDIA. 2025. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[94] NVIDIA. 2025. Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution. https://github.com/triton-inference-server.

[95] NVIDIA and Victor Podlozhnyuk. 2013. Black-Scholes option pricing. https://github.com/NVIDIA/cuda-samples/blob/master/Samples/5_Domain_Specific/BlackScholes/doc/BlackScholes.pdf.

[96] Sung-Heun Oh and Seung-Min Yang. 1998. A Modified Least-Laxity-First scheduling algorithm for real-time tasks. In *Proceedings Fifth International Conference on Real-Time Computing Systems and Applications*. 31–36. https://doi.org/10.1109/RTCSA.1998.726348

[97] Young H. Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W. Lee. 2021. Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. 584–597.

[98] OpenXLA. 2025. XLA: Optimizing Compiler for Machine Learning. https://openxla.org/xla/tf2xla.

[99] Nathan Otterness and James H. Anderson. 2021. Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*. New York, NY, USA, 24–34. https://doi.org/10.1145/3453417.3453432

[100] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[101] J. Park, Yongjun Park, and S. Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[102] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat T. Chakradhar. 2011. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the IEEE International Symposium on High-Performance Parallel Distributed Computing*.

[103] Christopher J. Rossbach, Jon C. Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*.

[104] Kyle Rupnow, Wenyin Fu, and Katherine Compton. 2009. Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking. In *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines*. 63–70.

[105] SeoWonik, ChaSanghoon, KimYeonjae, HuhJaehyuk, and ParkJongse. 2021. SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms. *ACM Transactions on Architecture and Code Optimization* 18 (2021), 1–26.

[106] Amna Shahid and Malaika Mushtaq. 2020. A Survey Comparing Specialized Hardware And Evolution In TPUs For Neural Networks. In *Proceedings of the 23rd IEEE International Multitopic Conference*. 1–6.

[107] Lin Shi, Hao Chen, and Jianhua Sun. 2009. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2009.5161020

[108] Roy Spliet and Robert D. Mullins. 2018. The case for limited-preemptive scheduling in GPUs for real-time systems. In *Proceedings of the Operating Systems Platforms for Embedded Real-Time Applications*.

[109] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*. New York, NY, USA, 1075–1092. https://doi.org/10.1145/3627703.3629578

[110] Hsin-Hsuan Sung, Yuanchao Xu, Jiexiong Guan, Wei Niu, Bin Ren, Yanzhi Wang, Shaoshan Liu, and Xipeng Shen. 2022. Brief Industry Paper: Enabling Level-4 Autonomous Driving on a Single $1k Off-the-Shelf Card. In *Proceedings of the 28th IEEE Real-Time and Embedded Technology and Applications Symposium*. 297–300.

[111] Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. 2017. GLoop: an event-driven runtime for consolidating GPGPU applications. In *Proceedings of the 2017 Symposium on Cloud Computing*.

[112] Tianxiang Tan and Guohong Cao. 2024. Thermal-Aware Scheduling for Deep Learning on Mobile Devices With NPU. *IEEE Transactions on Mobile Computing* (2024).

[113] I. Tanasić, Isaac Gelado, Javier Cabezas, A. Ramírez, N. Navarro, and M. Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture*. 193–204.

[114] Andrew S. Tanenbaum. 2007. *Modern Operating Systems* (3rd ed.). Prentice Hall Press, USA.

[115] TKEStack. 2022. GaiaGPU: vcuda-controller. https://github.com/tkestack/vcuda-controller.

[116] Linus Torvalds and Intel. 2025. Linux kernel – ivpu driver. https://github.com/torvalds/linux/tree/master/drivers/accel/ivpu.

[117] A. Tsutsui, T. Miyazaki, K. Yamada, and N. Ohta. 1995. Special purpose FPGA for high-speed digital telecommunication systems. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors*. 486–491. https://doi.org/10.1109/ICCD.1995.528912

[118] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA, 372–383. https://doi.org/10.1145/3352460.3358307

[119] Guibin Wang, Yisong Lin, and Wei Yi. 2010. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *Proceedings of the IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. 344–350.

[120] Hongqiang Wang, Jay Yun, and Alex Bourd. 2018. OpenCL Optimization and Best Practices for Qualcomm Adreno GPUs. In *Proceedings of the International Workshop on OpenCL*. New York, NY, USA, Article 16, 8 pages. https://doi.org/10.1145/3204919.3204935

[121] Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen. 2025. Colocating ML Inference and Training with Fast GPU Memory Handover. In *Proceedings of the 2025 USENIX Annual Technical Conference*.

[122] Tianxia Wang, Zhuofu Chen, Xingda Wei, Jinyu Gu, Rong Chen, and Haibo Chen. 2024. Characterizing Network Requirements for GPU API Remoting in AI Applications. arXiv:cs.OS/2401.13354 https://arxiv.org/abs/2401.13354

[123] Yidi Wang, Cong Liu, Daniel Wong, and Hyoseung Kim. 2024. Unleashing the Power of Preemptive Priority-based Scheduling for Real-Time GPU Tasks. arXiv:cs.DC/2401.16529 https://arxiv.org/abs/2401.16529

[124] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*. 358–369. https://doi.org/10.1109/HPCA.2016.7446078

[125] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of Service Support for Fine-Grained Sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. New York, NY, USA, 269–281. https://doi.org/10.1145/3079856.3080203

[126] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[127] Wikipedia. 2025. List of AMD graphics processing units. https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units.

[128] Wikipedia. 2025. List of Nvidia graphics processing units. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units.

[129] Bo Wu, Xu Liu, Xiaobo Zhou, and C. Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[130] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

[131] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao

[132] Xilinx. 2018. AWS F1 Xilinx Developer Labs. https://github.com/Xilinx/AWS-F1-Developer-Labs/tree/master/helloworld_ocl.

[133] Yu Xue, Yiqi Liu, Lifeng Nai, and Jian Huang. 2023. V10: Hardware-Assisted NPU Multi-tenancy for Improved Resource Utilization and Fairness. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

[134] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F. Donelson Smith, James H. Anderson, and Jan-Michael Frahm. 2019. Re-Thinking CNN Frameworks for Time-Sensitive Autonomous-Driving Applications: Addressing an Industrial Challenge. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 305–317.

[135] T. Yeh, Matthew D. Sinclair, Bradford M. Beckmann, and Timothy G. Rogers. 2021. Deadline-Aware Offloading for High-Throughput Accelerators. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 479–492.

[136] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chi-Yuan Chou. 2020. KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*.

[137] Jiaqi Yin, Zhiru Zhang, and Cunxi Yu. 2022. Exact Memory- and Communication-aware Scheduling of DNNs on Pipelined Edge TPUs. In *Proceedings of the 7th IEEE/ACM Symposium on Edge Computing*. 203–215.

[138] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. 2022. FAERY: An FPGA-accelerated Embedding-based Retrieval System. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. Carlsbad, CA, 841–856. https://www.usenix.org/conference/osdi22/presentation/zeng

[139] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

[140] H. Zhou, G. Tong, and Cong Liu. 2015. GPES: a preemptive execution system for GPGPU computing. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 87–97.

A Artifact Appendix

This artifact provides the source code of XSched, a detailed readme, and scripts to reproduce the main experimental results from the OSDI 2025 paper—"XSched: Preemptive Scheduling for Diverse XPUs" by W. Shen, M. Han, J. Liu, R. Chen, and H. Chen. XSched is a scheduling framework that enables preemptive scheduling on diverse XPUs through a general XQueue abstraction and multi-level hardware model. We provide instructions to build the software package and run experiments. Our artifact obtained the "Artifacts Available," "Artifacts Functional," and "Results Reproduced" badges from the Artifact Evaluation process of OSDI 2025. The DOI of our artifact is https://doi.org/10.5281/zenodo.15308935.

Artifact repository. The project source code and comprehensive instructions for building and running the main experiments on different XPUs are available in this GitHub repository: https://github.com/XpuOS/xsched-artifacts.git



OS Rendering Service Made Parallel With Out-of-Order Execution and In-Order Commit

Yuanpei Wu[1,2] Dong Du[1,2] Chao Xu[3] Yubin Xia[1,2]
Yang Yu[1,2] Ming Fu[3] Binyu Zang[1,2] Haibo Chen[1,2]

[1]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
[2]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education*
[3]*Fields Lab, Huawei Central Software Institute*

Abstract

Rendering service is an indispensable OS service on smart-device OSes like Android, iOS and OpenHarmony. However, the recent shift towards highly scalable display scenarios, such as foldable and multiple screens, has notably amplified the rendering workload, leading to low frame rates that degrade user experience. Yet, rendering services predominantly follow a sequential model, which is notoriously hard to parallelize due to the complex *state dependency*, *drawing order dependency*, and *interface dependency*.

This paper observes that a significant portion of the rendering procedure is potentially parallelizable through proper state pre-untangling and drawing order post-preserving. To this end, this paper introduces *Spars*, a scalable parallelized OS rendering service inspired by the out-of-order execution with in-order commit in computer architecture. *Spars* revolutionizes the rendering procedure by initially generating self-contained rendering tasks through *in-order preparation*, executing such tasks in an *out-of-order manner* to maximize multi-core parallelism, and subsequently committing the tasks *in-order* to enforce drawing order dependencies. Evaluation results on state-of-the-art single-screen, dual-fold, and tri-fold smartphones (Mate 70, X5, XT) as well as one-chip-multiple-screen configurations show an average frame rate improvement of 1.76×–1.91×. Moreover, *Spars* is able to decrease the device power consumption by 3.0% or increase the budget of graphics primitives by 2.31× for more appealing visual effects with the same stable frame rate.

1 Introduction

Rendering service is a crucial system service on smart-device OSes like iOS [19], Android [6], and OpenHarmony [22], enabling the display of exquisite graphical user interfaces (GUIs). On average, people worldwide spend 6 hours and 40 minutes per day on screens [56], not accounting for emerging graphical devices such as intelligent cockpits, head-up displays, smart homes, and wearable devices. Rendering service represents one of the most demanding workloads in smart OSes — our evaluations show that rendering-related tasks typically occupy 65%–95% of total CPU and GPU running time in commercial smartphones.

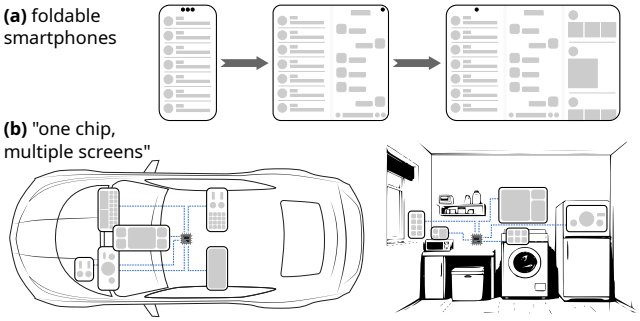


Figure 1: Demands for rendering scalability in foldable smartphones and one-chip-multiple-screen display setups.

The workflow of state-of-the-art rendering services follows a *sequential* model, encompassing management of a render tree, translation of draw commands into GPU objects, and GPU rasterization into pixels. First, the rendering service maintains a render tree, where each render node stores draw commands and parameters of graphics primitives, such as the height and width of a rectangle. Meanwhile, each node contains relative information with respect to its parent, such as the relative position, and its child nodes are sorted based on the drawing order from back to front. Subsequently, a 2D drawing engine translates draw commands in the tree into GPU objects like meshes, textures, and pipelines. When sequentially traversing the tree node by node, it tracks a stack of relative information and enforces the correct drawing order. Eventually, the GPU handles triangle rasterization, texture sampling, pixel shading, etc., based on the GPU objects provided through standard GPU APIs like Vulkan [34].

Emerging demand for scalable rendering. While the sequential model continues to be effective for traditional devices, the recent shift towards smart devices featuring foldable screens [11, 13, 16, 17, 25] and multiple screens [7, 46] in smartphones and intelligent cockpits (Figure 1) has placed increasing pressure on the OS rendering services. For example, while the Huawei Mate 70 (single-screen), Mate X5 (dual-fold), and Mate XT (tri-fold) possess similar System-

on-Chip (SoC), the latter two need to render 70% and 117% more pixels in each frame due to the larger screen area. The limitations of state-of-the-art sequential rendering services in coping with such heightened demands compel smartphone manufacturers to make compromises on other display parameters, such as pixel density reduction in Samsung and frame rate decrease in Huawei (§3.1). Intelligent cockpits with one-chip-multiple-screen display configurations face the same dilemma of trade-offs.

Key observation: bottleneck due to core underutilization in rendering services. We have identified a critical issue of underutilization in the hardware capabilities of rendering services. Despite the availability of multiple CPU cores, many remain underutilized or idle, even when they could contribute to handling the increased workloads. In real-world experiments (§3.3), for example, the rendering service's render thread monopolizes 80% utilization of a single core, limiting the frame rate. The remaining cores are only partially engaged in application logic, leaving 9 out of 12 cores idle and underutilized. This underutilization highlights the potential to improve performance by better leveraging the multi-core parallelism of modern SoCs, enabling rendering services to scale more effectively to meet the growing demands.

Based on this key observation, we argue that *the next-generation rendering service should adopt a parallel architecture to address scalable rendering demands*. Nevertheless, designing a scalable and parallel rendering service presents three significant challenges posed by dependencies.

Challenge-1 (C1): State dependency. Rendering tasks exhibit state dependencies that hinder parallelization efforts. Each node only contains the relative information with its parent. This design ensures that modifying a single node triggers updates across its entire subtree, while necessitating a sequential depth-first traversal to convert the relative information into absolute values before executing each draw command.

Challenge-2 (C2): Drawing order dependency. Rendering tasks involve critical drawing order dependencies that must be maintained. When two graphics primitives overlap, the background should always be rendered before the foreground (i.e., *overlapping relations*). While the render tree organizes the children of each node from back to front and preserves the sequence through a depth-first traversal, the pre-defined drawing order impedes parallelization efforts.

Challenge-3 (C3): Interface dependency. Refactoring for parallelization would impact existing interface compatibility. Traditional 2D rendering systems utilize a stateful interface, such as Skia Canvas [27] or Drawing API [9], where new commands may rely on past commands. Also, traditional state-based rendering optimizations like command batching should not be ignored under the new rendering paradigm.

To tackle these challenges, we introduce *Spars*, a scalable and parallel OS rendering service, powered by a novel drawing engine *Spade*2D. Our key insight in overcoming the aforementioned challenges is that *rendering tasks do not need to be executed sequentially to satisfy the three dependencies* — our study reveals that a large portion (76%) of the rendering procedure could be self-contained and potentially parallelizable, as long as *the states are properly untangled for inputs* and *the drawing orders are carefully maintained for outputs*. This approach is closely analogous to out-of-order execution in computer architecture [83], where instructions are dispatched as soon as dependencies are resolved, while their results are committed in-order to ensure program correctness.

Drawing inspiration from the concept, *Spars* employs a three-stage rendering procedure: *in-order preparation*, *out-of-order execution*, and *in-order commit*, to effectively utilize multi-core SoCs without violating dependency constraints. Specifically, *Spars* introduces the concept of *self-contained rendering tasks* derived from the render tree, allowing each task to be executed out-of-order and in parallel, thereby untangling state dependencies (**C1**). We observed that the actual execution of rendering does not impact the preparation of required drawing states. A quick in-order dry run, without rendering, suffices to decouple non-parallelizable states from the parallelizable tasks. Next, to ensure correct drawing order (**C2**), *Spars* explicitly manages overlapping relations of the rendering tasks for a dedicated commit thread. A finished rendering task can be committed to the GPU command only if all the drawing order dependencies are respected. Finally, *Spars* decouples the rendering procedure into two phases: a stateful phase and a stateless phase. The former maintains a consistent interface (**C3**) for custom-rendering applications and state-based optimizations, while the underlying *Spade*2D drawing engine, in contrast, employs a stateless interface for parallelism and scalability.

We have implemented and tested *Spars*[1] with 42 representative smartphone scenarios on Mate 70, Mate X5, Mate XT, and 12 one-chip-multiple-screen configurations. Comparing the results with the state-of-the-art sequential rendering models of OpenHarmony demonstrates an average frame rate increase of 1.76×–1.91×. Through leveraging multi-core parallelization, *Spars* is able to reduce the whole-device power consumption by 3.0% or allow a 2.31× budget for the total number of rendering graphics primitives for more appealing visual effects under the same frame rate. We anticipate that the design of *Spars* will significantly influence the evolution of next-generation OS rendering services.

2 OS Rendering Service Explained

The OS rendering service, as a system service, is responsible for rendering all of the GUI elements specified and synchronized from applications into every pixel (color value) of the frame buffer. This section uses the state-of-the-art iOS [31] and OpenHarmony [3] designs to illustrate the core rendering

[1] https://github.com/SJTU-IPADS/Spars-artifacts

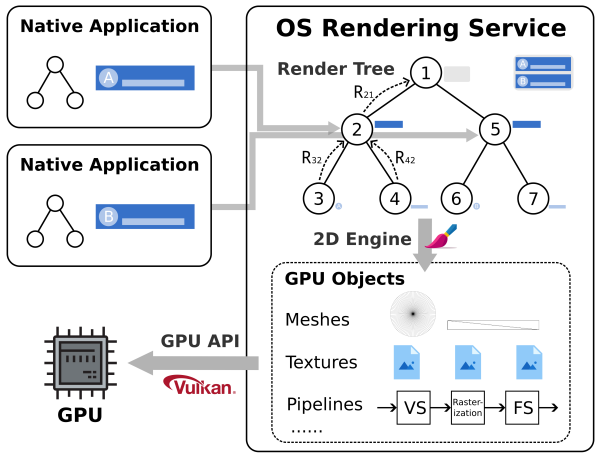


Figure 2: OS core rendering procedure.

procedure, demonstrated in Figure 2.

2.1 Render Tree

The render tree in the OS rendering service unifies and manages all of the drawing information on the screen(s). Different types of the render nodes on the tree record screen parameters, window parameters, and most importantly, parameters of the draw commands. Each draw command encodes a 2D graphics primitive, such as the height and width of a rectangle or the radius of a circle. When an application needs to update its content, it synchronizes a modified subtree or specific animation instructions of the render nodes with the OS rendering service through inter-process communication. Compared to the window-divided rendering approach used by Android [14] or Wayland [58], where each application process renders its own frame buffer locally for later system-wide composition, a unified render tree enables richer inter-window animations and visual effects. With finer-grained occlusion culling and reduced frame buffer memory usage, iOS [31], visionOS [30], OpenHarmony [3], etc., all adopt such strategy.

Implication-1: Current rendering services only maintain relative information in render nodes. Each render node on the tree only contains the *relative information* (R) with respect to its parent, such as the relative transform matrix (encoding translation, scaling, and rotation) and the relative clippings (scissoring the drawing region). This design ensures that modifying R of any node sufficiently and efficiently updates the entire subtree — for example, moving its position.

Implication-2: Drawing order should be strictly enforced for correctness. Each render node has its children sorted based on the *drawing order*, from back to front. Note that node drawing is not always commutative due to possible overlapping. Left subtree must be drawn before the right subtree based on the pre-definition. Therefore, a depth-first traversal of the render tree implicitly keeps the drawing order, as the numerical example (from 1 to 7) shown in Figure 2.

2.2 2D Engine

The 2D drawing engine translates the parameters of the draw commands (i.e., the mathematical expressions of 2D graphics primitives) into GPU objects like meshes, textures, and pipelines, which are later sent to GPU for actual rasterization.

Implication-3: The core rendering logic (translation from commands to GPU objects) is a time-consuming CPU procedure. Modern 2D engines like Skia [29] or Impeller [18] mostly use CPU tessellation to triangulate 2D primitives into meshes for GPU to rasterize, combined with signed distance field (SDF) [52, 70] and stencil then cover (StC) [61] algorithms for anti-aliasing and rendering specific graphics primitives. While SDF and StC help reduce some CPU load, the end-to-end rendering procedure still involves CPU-intensive tessellation, memory buffer filling, and other GPU object preparations, with CPU time accounting for 82% of the total frame rendering time in commercial OS rendering (§3.3). Besides, glyphs (text) are still rasterized by the CPU into an atlas using tools like FreeType [12] for functional completeness, and then mapped by the GPU to specific positions.

Implication-4: Existing 2D engines expose stateful interfaces for rendering, which keep internal states so past commands would affect future commands. When the rendering service depth-first traverses the render tree, it issues state-update commands to the underlying 2D engine using the relative information R. At any node during the traversal, the 2D engine always keeps a stack of relative information along the path — from the tree root to the current node — so that a subsequent draw command can derive the absolute state of the transform matrix and clippings.

Implication-5: Stateful interfaces enable 2D engines to use state-based optimizations internally. One important optimization is *command batching*. Rather than individually doing the translation one command by another, 2D engines first organize them into a chain structure, serving as an intermediate representation (IR) for GPU object generation. Each new command will be appended to the end of the chain, and then try to coalesce with the previous command if they can share the same GPU pipeline, i.e., using the same shaders and pipeline configurations. If not, the iteration can go one command backward as long as the current non-coalescible command does not overlap with the new command, so the drawing order is still correct if batching with previous commands. The backward iteration typically has a constant maximum limit to prevent the linear complexity from degrading into quadratic for building the entire chain.

2.3 GPU API

The GPU API, implemented by the GPU driver, provides standardized interfaces (i.e., abstractions of the GPU objects) that enable 2D engines to leverage the GPU for efficient graphics rasterization. It provides the capabilities of drawing triangles,

sampling textures (images), shading graphics primitives with various visual effects via programmable shaders, etc.

Implication-6: Modern GPU APIs are already stateless and can support parallel tasks. The next-generation GPU APIs such as Vulkan [34], Metal [20], and DirectX12 [8], on which state-of-the-art rendering services and 2D engines are built, are *stateless* within the GPU driver. In contrast with traditional APIs like OpenGL [23] which maintain a single-threaded state machine model within the GPU driver, they delegate the management of GPU objects to the API users. This provides a necessary condition for scalable and parallel rendering in multi-core configurations.

Implication-7: Modern GPU APIs still have constraints that must be considered in system design for compatibility. Modern GPU APIs still have *parallelization constraints*, considering the need for compatibility with existing GPU hardware. For instance, in Vulkan, command buffers should not be arbitrarily small because of the limitations in CPU submission and GPU scheduling. Also, parallel-recorded secondary command buffers cannot target at a different frame buffer (i.e., render pass) than the primary command buffer [59], which is a common case in OS rendering services where some components need to render into a separate layer. Our system is designed with these parallelization constraints in mind.

3 The Need for Scalable Rendering

In recent years, the demand for visually captivating, smooth, and immersive displays has been driving the evolution of next-generation smart devices [66, 87, 93], particularly in the form of foldable smartphones and multi-screen environments. Achieving larger display area, higher pixel density, and higher refresh rate at the same time has become both a user expectation and a technical challenge, pushing traditional fixed-thread sequential rendering models into their limits. This section analyzes the current and future display capabilities of flagship commercial smart devices, identifies their performance bottlenecks, and presents our observations.

3.1 Increased Rendering Loads in the Wild

Scenario-1: Foldable large screen. Foldable smartphones, including dual- [13, 16, 25] and emerging tri-fold devices [17], represent a significant advancement in screen design, and the market size was valued at 27.79 billion USD in 2023 with expected annual growth of 13.5% [11]. These devices offer users the ability and flexibility to view and interact with multiple pages and more content simultaneously, enhancing both immersion and productivity. However, the increased display area — compared to traditional single-screen devices — results in higher demands on frame rendering performance.

Figure 3 compares the screen specifications of contemporaneous flagship models. As foldable devices often use the same chipset as their single-screen counterparts, the larger display area in reality requires trade-offs in other aspects. For example, Samsung opts to reduce pixel density, keeping the total number of pixels similar to that of traditional smartphones with a sacrifice of definition. On the other hand, Huawei reduces the frame rate of foldable smartphones to 90Hz in common use cases, sparing more time for each frame to be rendered, even if today's mainstream screen hardware supports up to 120Hz, with a sacrifice of smoothness.

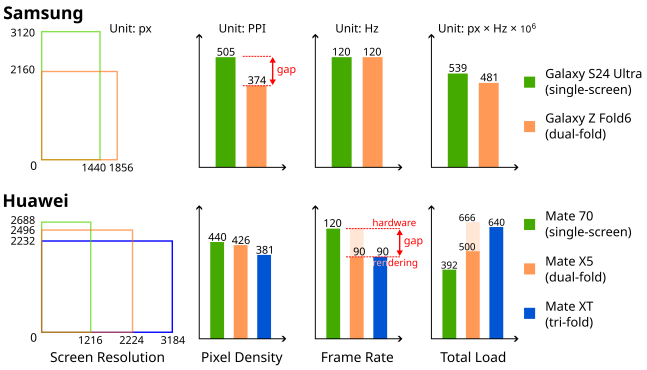


Figure 3: Display specifications of flagship smartphones. Large screen area means trade-offs in other parameters.

Scenario-2: One chip, multiple screens. In 2023, multi-screen solutions were installed in about 3.6 million cars in China, with 43 and 52 brands offering multi- or dual-screen intelligent cockpits, respectively [7]. For example, AITO M9 (2024) [5] features five screens: a center screen (1× 15.6"), a dashboard screen (1× 12.3"), a copilot screen (1× 16"), and two backseat screens (2× 17.3"). In multi-display environments, a single chipset is often responsible for driving multiple screens simultaneously to achieve a consistent and synchronous experience, with graphical elements able to freely and seamlessly stream between different screens [46].

Challenges arise for the OS rendering service when preparing content for multiple displays at once, as the computational demands increase proportionally. As a result, current state-of-the-art multi-screen environments can only operate at 45Hz–60Hz frame rate [92] and trade display quality (e.g., number of graphics primitives) for stable performance. This disparity between hardware display limit and software rendering capability underscores the need for a scalable and parallel OS rendering service.

3.2 State-of-the-Art Efforts

State-of-the-art rendering services in commercial operating systems adopt two classes of parallelization: *inter-frame parallelism* and *multi-window parallelism*, to tackle the CPU performance bottleneck in 2D rendering.

Inter-frame parallelism. iOS [31], Android [14], and OpenHarmony [3] all employ a multi-stage pipeline for inter-frame parallelism, where each frame is sequentially rendered across

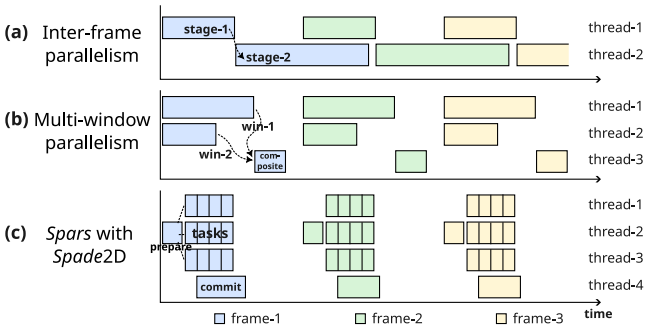


Figure 4: The threading model of different methods. Compared to state-of-the-art efforts (inter-frame and multi-window parallelism), our solution enables finer-grained parallelism.

a fixed set of threads, with consecutive frames in different stages rendered in parallel, as shown in Figure 4(a). Inter-frame parallelism does improve multi-core utilization, but it has three limitations. First, the number of threads that can be used are not scalable, as the number of stages (usually 2 or 3) are tightly coupled with the rendering logic, statically divided in advance. Second, the performance is bottlenecked by the longest stage. Due to the static logic-coupled division, stage imbalance is unavoidable. In OpenHarmony, we observed that the running time of stage-1 is <50% shorter than the heavily-loaded stage-2. Last, inter-frame parallelism may increase rendering latency, as each frame needs to span two to three Vertical Synchronization (VSync) periods [33] to finish.

Multi-window parallelism. In multi-window parallelism, each window is rendered in different threads (or processes), and later all the window frame buffers are composited into a single display frame buffer, as shown in Figure 4(b). Android [14] adopts this approach, and Skia Graphite [85] supports parallelization with render passes (i.e., tile or frame buffers). Although multi-window parallelism is useful for multiple windows, it also has three limitations. First, the workloads are highly imbalanced, as usually a single focused application occupies the entire workloads, with status or navigation bar at the corner seldom updating. Second, many scenarios do not have the windowing abstraction, which is however necessary for parallelism. For example, a chatting app on a large foldable screen can display the contacts, chat history, and posted photos simultaneously on different sections of the same window (Figure 9(d)), while the rendering service does not have any semantics. Last, multi-window parallelism introduces more memory for window frame buffers and additional workloads for window composition.

State-of-the-art system research. Existing system research usually focuses on multi-GPU parallelism for 3D rendering on screen walls [53, 62, 71, 80] and graphics emulators [51, 67, 68, 79], which is different from our smart device configurations and demands (i.e., OS 2D rendering). *D-VSync* [87] exploits single-core CPU capabilities in smartphones to tackle workload fluctuations, handling sporadic frame drops via pre-rendering during animations. Nonetheless, *D-VSync* cannot scale to large multiple screens and multiple CPU cores. Table 1 summarizes the system techniques for rendering performance.

Table 1: Performance comparison of different system techniques. Inter-frame and multi-window parallelism methods have been adopted by commercial smartphone OSes including Android, iOS, and OpenHarmony, but still cannot handle workload fluctuation and constant heavy loads due to the coarse-grained parallelism. *D-VSync* [87] is the state-of-the-art deployed in HarmonyOS NEXT, but cannot handle constant heavy loads due to its lack of parallelism support.

	Applicable Scenarios	Extra Loads	Rendering Latency	Workload Fluctuation	Constant Heavy Loads
Baseline	100%	/	/	Frame Drops	Low Frame Rate
Inter-frame Parallelism	100%	Minor	Serious Influence	Frame Drops	Medium Frame Rate
Multi-win Parallelism	<5%	Some	Some Influence	Frame Drops	Medium Frame Rate
D-VSync	85%	Minor	No Influence	No Frame Drop	Low Frame Rate
Spars with *Spade*2D	100%	Minor	No Influence	Leveraging *D-VSync*	High Frame Rate

State-of-the-art graphics research. Existing graphics research proposes various rendering algorithms that attempt to offload CPU-intensive 2D rendering onto GPUs and accelerators [36]. Legacy systems such as Windows GDI [86] and early versions of Cairo [75] and Skia [29] rely on pure CPU-based scanline rasterization [88]. Nowadays, modern 2D engines on smart devices primarily use tessellation and triangulation with SDF algorithms [52, 70] for GPU rasterization. A series of subsequent studies further explore stencil-buffer-based [61, 63, 64, 82] and vector-texture-based [50, 74, 78] algorithms. However, they increase the total load and power consumption on mobile (tile-based [2]) GPUs due to costly stencil operations and complex shader computation, limiting their universal applicability in OS rendering.

3.3 The Breakdown of Rendering Workloads

We present two quantitative analyses of rendering workloads based on our long-term industry experience with the OS rendering service.

Observation-1: CPU dominates the rendering costs. The end-to-end rendering process is CPU-intensive for 2D scenarios. Figure 5(a) illustrates the distribution of CPU and GPU execution time for each frame in representative scenarios, such as desktop and application pages. On average, sequential CPU processing accounts for 82% of the end-to-end frame rendering time. In contrast to 3D games that require the GPU to render millions of triangles per frame [72], 2D scenes typically involve fewer than 1,000 triangles. The primary workload of the OS rendering service falls on the CPU, responsible for preparing render tree states, tessellating and

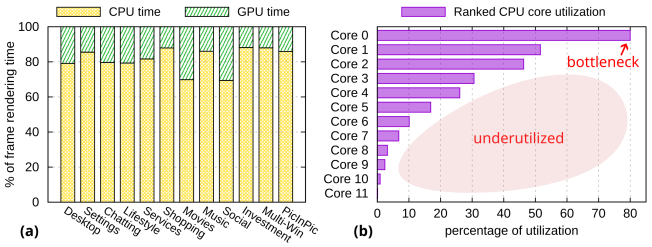


Figure 5: Breakdowns of rendering workloads on Huawei Mate X5, between (a) CPU and GPU, and (b) CPU cores.

triangulating mathematical abstractions of 2D primitives into vertices, and generating GPU objects. As the CPU and GPU run asynchronously, the CPU execution time fully covers the GPU execution time, becoming the bottleneck.

Observation-2: Sequential rendering model limits the hardware utilization. The current state-of-the-art rendering procedure follows a fixed-thread sequential model, restricting the multi-core utilization. Figure 5(b) highlights the imbalanced CPU per-core utilization in the Lifestyle scenario on Mate X5 (Figure 9(c)) with OpenHarmony 5.0, struggling to achieve the frame rate of 90Hz. While the rendering service's render thread monopolizes 80% utilization of a single core with inter-frame parallelism and application logic partially utilizing other cores, most CPU cores (cores 3∼11) remain largely idle, making the single core the performance bottleneck. Modern chipsets typically feature 8 to 12 CPU cores, with this trend set to continue [21]. Given the significance of rendering services in smart-device operating systems, achieving scalability and parallelization across multiple cores is crucial for optimal performance.

4 Scalable Rendering

4.1 Parallelism with Out-of-Order Execution

A multi-core parallel OS rendering service is the key to support scalable rendering. However, it faces three major challenges detailed in §1. First, *state dependency* prevents encapsulating tasks for parallel execution, as each node in the render tree only contains the relative information in terms of its parent. Second, *drawing order dependency* for graphics primitives must be maintained for correctness, while individual tasks scheduled on the worker threads lose the pre-defined order. Last, *interface dependency (compatibility)* should be considered for existing applications and optimizations that rely on stateful rendering, which contrasts with the stateless nature of parallelization.

Revisiting parallelism in computer architecture. In processors, parallelism can be achieved through out-of-order execution and in-order commit [83]. Out-of-order execution refers to a strategy where a processor executes instructions in an order different from their original program sequence. This technique allows the processor to *bypass potential instruction dependencies* and better utilize idle execution units, thereby enhancing overall performance and efficiency. On the other hand, in-order commit ensures that the results of these out-of-order executed instructions are committed to memory or storage in the original sequential order of the program, preserving the correctness and consistency of the final output.

Insights on parallel rendering. We observed that the challenges of parallel rendering exhibit similarities to those encountered in out-of-order processors, for example, the dependencies between draw commands and between instructions. Drawing inspiration from practices in computer architecture, our key insight is to *methodically decouple the rendering procedure into in-order task construction, out-of-order task execution, and in-order dependency enforcement*, enabling (mostly) parallel processing without violating dependency requirements. Specifically, the scalable and parallel rendering procedure comprises three stages: (1) *in-order preparation*, which involves constructing self-contained tasks based on the render tree, allowing out-of-order execution while ensuring the eventual enforcement of overlapping relations; (2) *out-of-order execution*, where tasks are executed using a configurable number of worker threads distributed across multiple cores; (3) *in-order commit*, which collects the execution results from the worker threads and serializes the outputs in a correct drawing order before submitting to the GPU for final rasterization. Compared to the design in computer architecture, a new stage, the in-order preparation, is introduced to satisfy specific requirements in parallel rendering.

The three-stage design increases minor total workloads, because it simply **decouples** the state management, drawing order management, and state-based optimizations — procedures originally existed and embedded in the state-of-the-art sequential rendering — from the core rendering workload, so the latter can be parallelized.

Building on the insight that scalability and parallelism can be achieved through out-of-order execution and in-order commit, this paper presents *Spars*, a novel scalable and parallel OS rendering service, powered by the underlying novel drawing engine *Spade*2D.

4.2 Design Overview

Figure 6 shows the overall rendering architecture of *Spars*. Instead of directly performing a depth-first traversal of the render tree to render each node sequentially, *Spars* divides the rendering procedure into the aforementioned three stages. These stages are supported by three types of threads in *Spars*: the *main thread*, *worker thread*, and *commit thread*.

In-order preparation (main thread). During the preparation stage, the render tree is still traversed once in a depth-first manner by the main thread to prepare self-contained rendering tasks (§5.1). The key difference is that no core rendering logic (i.e., invoking the 2D engine to generate any GPU object) is performed, making the in-order preparation a very fast dry

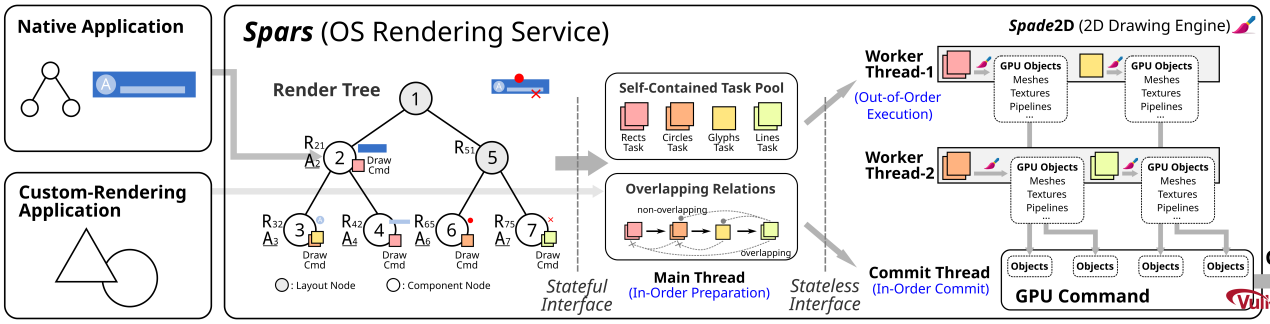


Figure 6: The parallel rendering architecture of *Spars* with *Spade*2D. The main thread prepares self-contained rendering tasks for out-of-order execution in the worker threads and overlapping relations for in-order commit in the commit thread.

run. At the same time, overlapping relations (§5.2) between the rendering tasks are recorded, which the commit thread will later use (in the in-order commit stage) to preserve a correct drawing order. State-based batching optimization (§5.3) can also be conducted here. Finally, the main thread puts all the self-contained rendering tasks into a single-producer multi-consumer (SPMC) task pool, and synchronizes the overlapping relations with the commit thread.

Out-of-order execution (worker thread). Worker threads in *Spars* execute rendering tasks retrieved from the SPMC pool. As every rendering task is self-contained without interdependency in states, they can be scheduled onto any worker thread with any order — enabling out-of-order and parallel execution. For each task, the worker thread invokes the *Spade*2D drawing engine to translate mathematical expressions of 2D graphics primitives into GPU objects like meshes, textures, pipelines, etc. *Spade*2D provides a stateless interface (§5.3), allowing each task to be executed independently on multiple cores without interference from one another (§5.4). Finally, the generated GPU objects are packed and put into a multi-producer single-consumer (MPSC) resource pool for the commit thread to perform in-order commit.

The number of worker threads is configurable at runtime, based on the chipset's multi-core configuration and the current execution environment. At least one worker thread is required. It is recommended to reserve several cores for the commit thread, application threads, and other system threads to avoid unnecessary scheduling overhead. In practice, 3 to 5 worker threads are typically sufficient (§6).

In-order commit (commit thread). The commit thread is responsible for gathering the generated GPU objects from the MPSC pool and committing them into the GPU command, based on the overlapping relations prepared by the main thread. This procedure is straightforward, with the commit thread simply binding the handles of the generated GPU objects into the GPU command by reference. Notably, the commit thread does not need to wait for all worker threads to complete their tasks. Instead, upon receiving a completed task with its associated GPU objects, the commit thread assesses the feasibility of committing it into the GPU command — whether all drawing-order-dependent tasks have already been committed (§5.2). If this condition is not met, the commit thread waits until all the dependent tasks complete and commit, ensuring a correct drawing order.

The waiting does not impact the end-to-end frame rendering time, as the main thread submits rendering tasks to the SPMC task pool in a naïve topological drawing order derived from a depth-first traversal. Although tasks are executed out-of-order with differing execution time, the completion order retrieved from the MPSC pool is mostly retained from the SPMC pool, and the chance of dependency waiting (i.e., waiting for the background primitive to complete first) in the commit thread is minor. Except for the last several rendering tasks, the execution time of the commit thread is fully covered by the worker threads, as illustrated in Figure 4(c). Finally, the GPU command will be submitted to the GPU driver for GPU rasterization, producing a color for each pixel.

5 Detailed Design

This section introduces detailed key techniques behind the design of *Spars* with *Spade*2D.

5.1 Self-Contained Tasks

For rendering to be scalable and parallel, *Spars* divides the entire rendering workloads into individual *self-contained* rendering tasks that can be executed on multiple worker threads in an out-of-order manner. This requires that self-contained tasks encapsulate all the essential drawing states for the *Spade*2D drawing engine to generate the corresponding GPU objects. A complete *drawing state* encompasses a transform matrix encoding the drawing position, scaling, and rotation, clippings encoding the valid drawing region, parameters encoding specific graphics primitives, as well as coloring and styles.

Challenge. However, as discussed earlier, rendering services face the challenges of state dependencies. In the existing render tree, each node does not contain the complete drawing state; instead, it only holds information relative to its parent. The benefit of the relative-state design is its efficiency in updating screen content, as modifying the root node of a subtree is sufficient to alter a group of nodes. In the render tree shown

in Figure 6, R_{ij} denotes the relative information of node i with respect to node j. For example, when an animation or a drag operation needs to move the card represented by the subtree rooted at node ②, it can just modify the transform matrix of R_{21}, which automatically moves node ③ and ④ with unchanged R_{32} and R_{42}, supposed that node ① is a larger container inside which node ② is able to move. The presence of state dependencies mandates that modern rendering services process render nodes (or tasks) sequentially in accordance with depth-first order to access the correct states, thereby impeding parallelization efforts.

Constructing self-contained tasks with a dry run. We observed that the actual execution of rendering tasks does not affect the construction of the complete drawing state required for them. Based on this observation, we introduce a *dry-run* method to generate the absolute information for each node. During the preparation stage, *Spars* traverses the render tree in a depth-first order (without invoking the 2D engine to do any actual rendering), calculates the complete states, and records the absolute information A_i — relative to the screen coordinates or the absolute zero — for node i that includes at least one draw command. By following this approach, *Spars* encapsulates the original draw commands in the render tree into a collection of self-contained rendering tasks devoid of state dependencies, as shown in Figure 6. When the screen updates in a frame, invalid A_i values are recalculated during the depth-first traversal in the main-thread preparation. For example, in a scenario where R_{21} undergoes updates (i.e., invalidation) due to drag operations, *Spars* dynamically adjusts the subtree containing nodes A_2, A_3, and A_4 during the dry-run traversal, before encapsulating tasks for these nodes. Notably, recorded A_6 and A_7 will not be affected, because their paths to the root have no relative information change.

Computation costs. The computation incurs minor additional workload overhead compared to the state-of-the-art rendering services. The conversion from relative to absolute information is originally integrated in the sequential rendering procedure through stateful interfaces provided by 2D engines. *Spars* simply decouples the fast, non-parallelizable preparation logic from the core parallelizable rendering logic. Additionally, unchanged absolute information is reusable across frames, making the preparation efficient and fast.

Resource costs. Storing absolute information incurs minor memory overhead. *Spars* observes the sparsity of draw commands relative to the total number of render nodes. Many internal nodes in the render tree are layouts rather than actual components, containing no draw commands. For example, node ① and ⑤ in Figure 6 only contain the relative information R, which does not require bookkeeping of any absolute information for drawing. In the desktop scenario on Mate XT, approximately 200 out of 800 nodes (25%) have draw commands, and each piece of absolute information, consisting of the transform matrix and clippings, occupies at most 10KB. Additional 2MB of memory usage is permissible.

5.2 Overlapping Relations

Rather than using the implicit drawing order pre-definition in the render tree, *Spars* explicitly manages the overlapping relations for in-order commit, which can enable out-of-order execution while not violating drawing order dependencies.

Challenge. Graphics rendering follows a partial order defined by the render tree, where tasks (i.e., graphics primitives) at the back must be drawn before tasks at the front when they overlap. For example, the dark blue background in Figure 6 must be drawn before the light blue and red components, while the order among node ③, ⑥, and ⑦ does not affect the correctness of drawing. Traditional sequential rendering applies a depth-first traversal when processing each node, as the children of every node are pre-defined to be in back-to-front order (i.e., z-order). However, the out-of-order task execution in *Spars* does not provide such a guarantee.

Overlapping relations. To tackle this challenge, *Spars* entrusts the main-thread preparation to explicitly construct overlapping relations of the self-contained rendering tasks for the commit-thread in-order commit. While it is possible to build a complete directed acyclic graph (DAG) for the partial order, *Spars* finds itself costly and unnecessary to do so — the commit thread just needs to follow one correct topological order, and the commit stage barely becomes a bottleneck as discussed in §4.2. Therefore, during the depth-first traversal in preparation, *Spars* just chains the generated self-contained tasks for a naïve drawing order and manages their axis-aligned bounding boxes (AABBs) [48]. An AABB is defined as the x- and y-axes aligned rectangular bounding box of each task's drawing region. It is one of the most efficient methods in computer graphics for determining whether two graphics primitives overlap — if two primitives overlap, their AABBs must also overlap. The chain with AABBs is later synchronized from the main thread to the commit thread.

In-order commit with overlapping relations. During the commit stage, the commit thread checks whether it is possible to commit the received completed task (i.e., its generated GPU objects) into the GPU command, based on the task chain and AABBs. If the task received is the current head of the chain when all the tasks before it have already been committed, then it is correct to commit the task next. This is the most common case, as the tasks (with possible varying execution time) are dispatched from the main thread to the worker threads using the same chain order. If the task is not the current head, then the commit thread checks the AABBs of all the unfinished tasks before it in the chain. When no AABB overlaps with the current completed task, then it is also correct to commit the task next into the GPU command. For example, in Figure 6, the *Lines Task* can be committed as long as the *Rects Task* has been committed. Otherwise, the commit thread must block to wait for a valid background task to commit first. The AABB checking only performs a preset maximum times (3 to 5 times) for each completed task. The

```
1 ▶Stateful APIs◀
2 /* state-update commands */
3 void Canvas::translate(float x, float y);
4 void Canvas::rotate(float degrees);
5 void Canvas::clip(Shape &shape);
6 void Canvas::save();
7 void Canvas::restore();
8 /* draw commands */
9 void Canvas::drawRect(Rect &rect);
10 void Canvas::drawCircle(Circle &circle);
```

```
11 ▶Stateless APIs◀
12 /* self-contained tasks */
13 struct RectsTask {
14   int primitiveCount;
15   Rect *rects; // a batch of rects
16   State *states; // the corresponding
     ↪ complete and absolute states
17 };
18 GpuObjects Spade2D::draw(RectsTask &task);
19 GpuObjects Spade2D::draw(CirclesTask &task);
```

Figure 7: Stateful and Stateless APIs in *Spars*.

mechanism ensures that one or two heavy tasks do not block the commits of subsequent tasks, so the commit procedure can almost be covered by the execution time of the heavily-loaded worker threads. Although there is no complete construction of a DAG, the commits follow a logical partial order, based on the overlapping relations of the chain and the AABBs.

Costs. The construction of the overlapping relations incurs minor extra workload, as traditional 2D engines use a similar chain structure for command batching (§2.2). In *Spars*, the same chain is reused for both the state-based batching (§5.3) and the overlapping relations. Moreover, pointer maintenance and AABB checks are lightweight operations.

5.3 Stateful and Stateless APIs

Existing rendering services and 2D engines utilize stateful APIs, hindering parallel rendering. Although it is possible to define new APIs capable to support parallel rendering without state dependencies, they are not compatible with custom-rendering apps and are not intuitive to use. Therefore, we decouple APIs into two parts: *stateful* and *stateless* APIs. The stateful APIs (provided by *Spars*) ensure compatibility with existing interfaces and support state-based batching optimizations, while the stateless APIs (provided by *Spade*2D) can enable parallel and out-of-order task execution. The vertical dashed lines in Figure 6 show the division.

Definitions. *Spars* defines *stateful* as a paradigm in which each executed command leaves corresponding states in the system, impacting the completeness and correctness of subsequent commands, as shown in Figure 7. This includes state dependencies, drawing order, and command reordering or coalescing. In contrast, a *stateless* interface implies that each rendering task is independent and self-contained, leaving no residual state that affects the correctness of subsequent draws.

Interface compatibility. The decoupling ensures interface compatibility with traditional 2D drawing engines for custom-rendering applications who might directly invoke them for drawing, such as games, Flutter apps [10], and web apps [90]. Although the underlying *Spade*2D is stateless, the stateful APIs serve as an adapter to untangle the intricacies and encapsulate self-contained tasks for it. Therefore, these third-party apps can enjoy the same traditional stateful interface such as the Drawing API [9] or Skia Canvas [27] for compatibility, ensuring practicality and applicability of our system.

Traditional state-based optimizations like command batching are also supported based on the decoupling. Draw commands that can share the same GPU pipeline prefer to be grouped together instead of processed individually, thus producing a single set of GPU objects and reducing the number of draw calls. This also saves CPU execution time for less GPU object management (§5.4) and fewer rendering tasks. The main-thread preparation does the same re-ordering and coalescing algorithm discussed in §2.2, when depth-first traversing the render tree and leveraging the same chain structure and AABBs for overlapping relations. For example, the two rectangle draw commands (shown in pink) in Figure 6 are batched together to form a single *Rects Task* for *Spade*2D GPU object generation. Since pointer operations and AABB checks suffice, the non-scalable stateful APIs decoupled from the parallelizable core rendering logic is lightweight and fast.

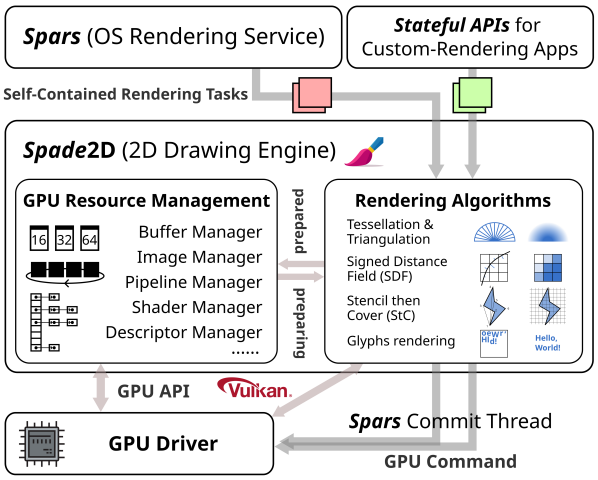


Figure 8: The architecture of *Spade*2D drawing engine.

5.4 *Spade*2D Drawing Engine

*Spade*2D is responsible for the stateless rendering (i.e., task out-of-order execution) in the worker threads. Similar to traditional drawing engines like Skia [29], Impeller [18], Direct2D [60], etc., *Spade*2D converts mathematical expressions of 2D graphics primitives into GPU objects for frame buffer rasterization. However, *Spade*2D differentiates itself in terms of *input and output formats* as well as *GPU resource management*. Figure 8 illustrates the architecture of *Spade*2D.

Input and output formats. Rather than accepting both state-update commands and draw commands, *Spade*2D employs stateless inputs: self-contained rendering tasks that can be executed out-of-order and in parallel across multiple CPU cores. *Spade*2D relies on external mechanisms to maintain state and drawing order dependencies. Furthermore, instead of handling a single primitive (e.g., a rectangle) per draw command, each rendering task in *Spars* may contain a batch of primitives (e.g., a vector of rectangles) that can share the same rendering algorithm and produce a single set of GPU objects. For outputs, *Spade*2D depends on the *Spars* commit thread for the in-order commit before submitting to the GPU,

for correct overlapping relations.

GPU resource management. GPU resources in *Spade*2D are managed by *thread-safe* managers that *prevent resource double creation* caused by parallel rendering. Similar to other 2D engines, *Spade*2D tracks GPU resources needed for various rendering algorithms, including buffers, images, pipelines, shaders, descriptors, etc. These resources are shared among rendering tasks and reused across frames if possible, each of which has a manager using size-class-based pools, hash tables, circular queues tied to the frame buffer rotation, etc.

Thread safety. In *Spade*2D, operations on these data structures need to be thread-safe. However, note that modern stateless GPU APIs, such as Vulkan [34], Metal [20], and DirectX12 [8], allow the parallelization of actual resource usage and creation within each rendering task. Time-consuming operations like vertex and index buffer filling, image decoding and creation, and GPU pipeline creation can all be executed without locks. Also, *Spade*2D uses shared locks [4] for resource reading, ensuring no contention when multiple rendering tasks refer to the same created GPU object (e.g., pipeline or image). Thus, the cost of synchronization is minimized and unnoticeable, as insertions and deletions for GPU resource handles in most data structures can finish in constant time.

Preventing resource double creation. Double creation might happen in parallel rendering when two rendering tasks refer to the same unprepared GPU resource simultaneously on different cores, and both attempt to prepare it, for example, drawing the same new image or using the same new GPU pipeline. To prevent double creation of GPU resources, *Spade*2D maintains *preparing* GPU resources in the resource managers, in addition to the *prepared* (created) and *unprepared* (free) GPU resources. The first task that needs a specific resource will atomically mark it as *preparing* in the corresponding manager before actually creating it. When the second task needs the same *preparing* resource, it will block and yield to another rendering task until being notified by the *prepared* condition variable. This mechanism avoids unnecessary double creation with no impact on performance, as blocking occurs only in rare cases when two similar non-batched tasks have contention, and after all other self-contained tasks can be scheduled to run.

6 Evaluation

6.1 Implementation and Setup

Implementation. We have implemented the scalable and parallel OS rendering service *Spars* with the stateless drawing engine *Spade*2D in C++ with the Vulkan GPU backend [34], the most common modern graphics API in smart devices. We deploy *Spars* as a custom-rendering service that directly utilizes GPU resources and bypasses the vanilla rendering service in Android and OpenHarmony for performance evaluation.

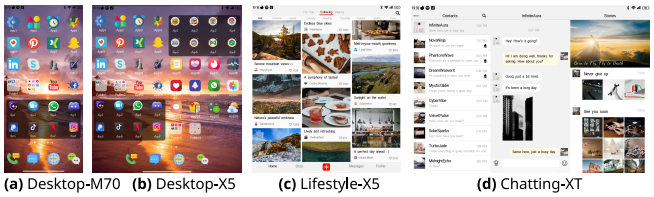


Figure 9: Representative scenarios rendered by *Spars*.

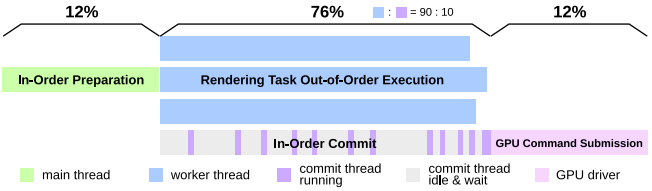


Figure 10: Execution time proportion of a frame in *Spars*.

We run *Spars* with *Spade*2D on state-of-the-art commercial smartphones, Mate 70 (single-screen) [15], Mate X5 (dual-fold) [16], and Mate XT (tri-fold) [17], as well as multi-screen configurations using the Kirin 9010 chipset [54].

Methodology. To analyze *Spars* performance under real scenarios, we export the render trees in OpenHarmony rendering services (in commercial smartphones) and reconstruct them in *Spars*. Figure 9 shows representative scenarios rendered by *Spars* with *Spade*2D. *Spars* is sound enough to render common graphics primitives, such as rectangles, rounded rectangles, circles, lines, images, glyphs (text), etc., with different coloring and styles. As *Spars* still lacks some features compared with vanilla rendering services, to better analyze the benefits and costs of our parallel design, we build *Sequential*, a sequential version of *Spars* with the standard rendering procedure discussed in §2. In addition to *Commercial*, the commercial rendering service in OpenHarmony 5.0, we also report the data of *Sequential* as another baseline for a more fair comparison with *Spars*.

In the following sections, we evaluate the parallelizable proportion (§6.2), frame rate improvement (§6.3), multi-core utilization (§6.4), power consumption reduction (§6.5), scalability (§6.6), and costs (§6.7) of *Spars*.

6.2 Theoretical Speedup Analysis

Spars enhances performance for rendering content that features a large number of rendering tasks, including diverse images, text, graphics primitives, and visual effects. The potential performance gains of *Spars* are determined by the parallelizable proportion of the end-to-end rendering workloads. As described by Amdahl's Law [39], the maximum achievable speedup is limited by the inverse of the fraction of the workload that must be executed sequentially [81].

Figure 10 illustrates the proportion of the execution time for different stages of a frame in *Spars*, averaged across all 42 scenarios (§6.3). The parallelizable out-of-order task exe-

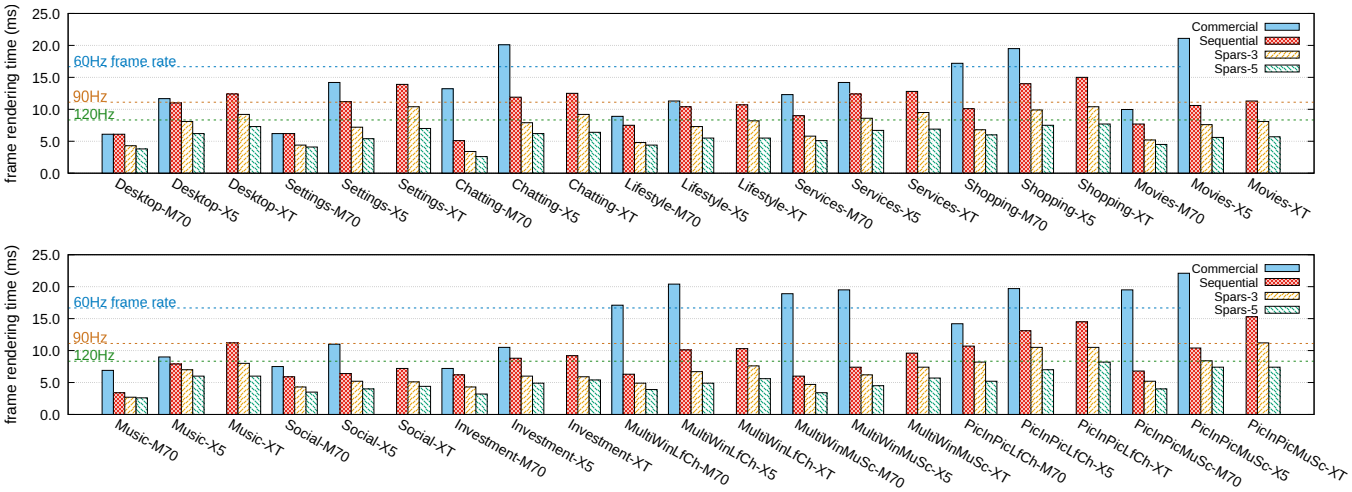


Figure 11: Frame rendering time on Mate 70, X5, XT with *Commercial*, *Sequential*, and *Spars* of 3 or 5 worker threads. All loads are pinned to medium cores with the same clock rate. Relative to *Sequential*, *Spars*-5 improves the frame rate by 1.76×.

cution and the in-order commit altogether account for 76% of the total workload. The commit thread is mostly idle and waits for the rendering tasks to complete. The non-parallelizable in-order preparation and the GPU driver command submission collectively occupy the remaining 24%.

With 3 worker threads, Amdahl's Law yields a theoretical maximum speedup of 2.14×, and with 5 workers, the speedup increases to 2.65×. In practice, however, scheduling and synchronization overhead during task dispatch, result collection, and resource management is unavoidable, influencing the overall performance.

6.3 Frame Rate Improvement

We evaluate the frame rate improvement of *Spars* on the latest Mate 70 (single-screen), Mate X5 (dual-fold), Mate XT (tri-fold), and Kirin 9010 chipset for multi-screen configurations. They all feature a heterogeneous CPU architecture with 4 little cores, 6 medium cores (3 physical cores with simultaneous multithreading [84]), and 2 big cores (1 physical core).

Common smartphone scenarios. We test *Spars* across altogether 42 representative smartphone use cases. Figure 9 shows some examples. These scenarios, drawn from our years of experience with OS rendering service deployment, provide a comprehensive snapshot of everyday smartphone usage — such as desktop operations, chatting, shopping, browsing services or movies — all of which involve rendering a variety of graphics primitives. We export and adapt the page layouts (i.e., render trees) from real-world apps and fill in dummy data for *Spars* to render. Each scenario is tested against two baselines, *Commercial*[2] and *Sequential*, and with *Spars* configured to use 3 and 5 worker threads (denoted *Spars*-3 and *Spars*-5), all pinned to medium cores running at the same clock rate (approximately 0.7GHz) as the baselines.

[2] *Commercial* data for Mate XT is not reported.

Figure 11 demonstrates the frame rate improvement. On average, *Spars*-3 reduces the CPU frame rendering time of *Sequential* by 27.3%, and *Spars*-5 reduces it by 43.2%. Consequently, the average frame rate could be increased by 1.38× and 1.76×, respectively. *Spars*-5, the standard configuration of *Spars* on these devices, achieves stable 120Hz frame rate across all 42 tested scenarios, whereas 27 (64%) sequential rendering baselines struggle to maintain the smooth experience, especially on large foldable screens. In heavily-loaded multi-window and picture-in-picture scenarios, *Spars*-5 is able to increase the frame rate by at most 2.07× with higher parallelizable proportion and better load balancing.

Worker thread configurations. *Spars* allows the number of worker threads to be configured both statically — based on the smart-device's SoC — and dynamically at runtime. On current SoCs with 6 medium cores, we use 5 worker threads as the standard setup, reserving one core for the commit thread. Dynamically adjusting the number of worker threads would require estimating or predicting the total workload, which is challenging due to the inherently dynamic nature of rendering [87] — for instance, whether certain GPU objects are already cached. In practice, we observed no performance regressions when increasing the number of worker threads, as long as the total remains below the number of available cores.

Core and clock rate configurations. A straightforward and traditional way for *Sequential* to boost the frame rate is to use a bigger CPU core or apply a higher clock frequency. However, this results in significantly more power consumption, often leading to unsustainable thermal output within minutes. As chip advancements slow with the decline of Moore's Law [73], *Spars* takes a different strategy by leveraging CPU multi-core parallelism to enhance the frame rate, offering long-term scalability across diverse (and possibly weak) smart-device hardware platforms.

To show the general benefits of *Spars*, we apply different

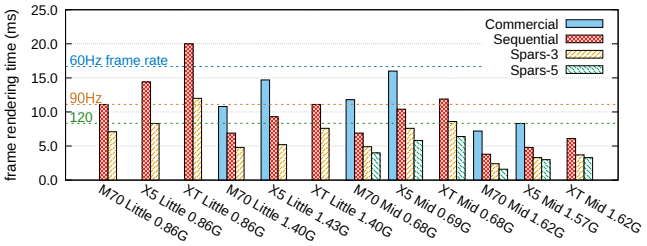


Figure 12: Frame rendering time with different configurations of cores and clock rates.

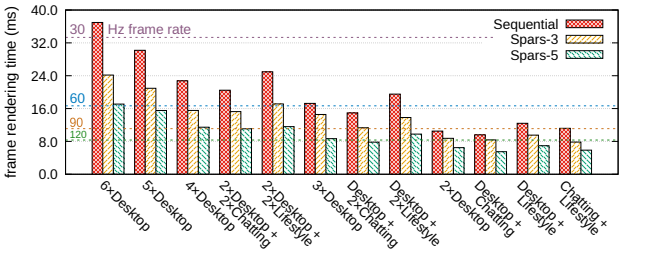


Figure 13: Frame rendering time with multiple screens.

configurations of CPU cores and clock rates on our smartphones. Specifically, we test *Commercial*[3], *Sequential*, and *Spars* on little cores and medium cores with both low and high clock rates. For a fair comparison with *Sequential*, we only report the frame rate improvement of *Spars* on homogeneous core configurations.

Figure 12 shows the average frame rendering time of all the scenarios under different configurations. For *Spars*-3 on little cores, it is reduced by 39.5% and 36.7% for low and high clock rates, respectively. Consequently, the frame rate is increased by 1.65× and 1.58×. For medium cores, *Spars*-3 brings 1.38× and 1.56× gains. Little cores have slightly more performance improvement, due to the lower synchronization overhead relative to the execution time. For *Spars*-5 on medium cores with low and high clock rates, the frame rate improvement is 1.76× and 1.89×.

One chip, multiple screens. We further run *Spars* to drive two to six 2K-resolution virtual screens on the Kirin 9010 chipset (2024), as shown in Figure 13. On average, *Spars*-3 boosts the frame rate by 1.34×, and *Spars*-5, the standard configuration, delivers 1.91×. Rendering more screens tends to have better performance gains. For six or five desktops, *Spars*-5 provides 2.16× and 1.94× gains, respectively, while for two desktops, it is just 1.62×. Heavier workloads yield a higher parallelizable proportion (more rendering tasks) in the end-to-end rendering procedure.

6.4 Multi-Core Utilization

The scalable and parallel rendering service optimizes the utilization of multi-core hardware in smart devices and ensures a more balanced workload across cores, preventing any single core from becoming the frame rate bottleneck.

[3] For reasonable frame rate, *Commercial* rendering service is never scheduled on little cores with low clock rate.

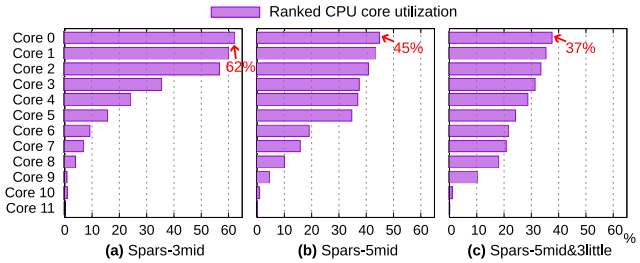


Figure 14: Multi-core CPU utilization of *Spars*.

Figure 14 demonstrates the system CPU utilization when *Spars* renders the Lifestyle scenario on Mate X5. The frame rate is locked at 90Hz, and the clock rate is kept the same for a fair comparison. As opposed to Figure 5(b) where the most loaded core reaches 80% utilization and bottlenecks the frame rate, *Spars*-3 and *Spars*-5 only lead to 62% and 45% utilization for the most loaded core. With a heterogeneous configuration of 5 medium cores and 3 little cores, the heaviest utilization further decreases to 37%, reducing the frame rendering time and potentially boosting the frame rate.

Influence on other workloads. *Spars* has minor influence on other system or application workloads, as it simply redistributes the entire rendering workload across multiple cores without increasing its total — additional logic for task encapsulation, dispatch, and collection is within 2%. As shown in Figure 14, all CPU cores still have a considerable amount of idle time available to process other tasks (although this idle time has been rebalanced). Other workloads are still scheduled and executed by the system scheduler according to their original priority.

6.5 Power Consumption Reduction

With balanced multi-core utilization, the power consumption of *Spars* is reduced in comparison to *Sequential* when running at the same configured frame rate.

The state-of-the-art QoS-guided scheduling and governing [43] puts the (parallelized) rendering workloads at little cores with lower clock rates if it predicts that the current frame can finish rendering within its VSync period (i.e., before its display deadline). Also, commercial DVFS (Dynamic Voltage and Frequency Scaling) can only adjust the frequency of an entire cluster (e.g., all medium cores) based on the QoS predictions. In *Sequential*, rendering on a single core often forces DVFS to raise the cluster voltage and frequency to meet the frame deadlines. In contrast, *Spars* spreads the workload across multiple cores, allowing DVFS to maintain the same stable frame rate at a lower overall voltage and frequency, and thus reducing the power consumption.

We measure the Mate XT whole-device instantaneous power consumption through battery counters [26] for 5 min-

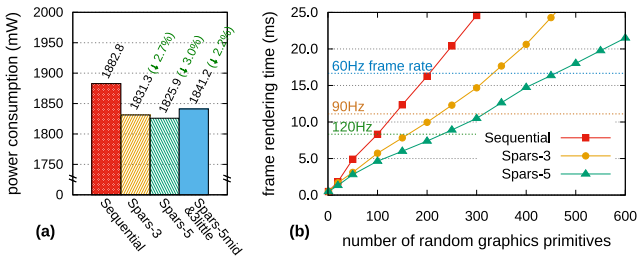


Figure 15: (a) Power consumption of *Sequential* and *Spars* under the same frame rate. (b) Scalability of *Spars*, allowing more graphics primitives under the same time budget.

utes after a one-minute warmup, with all other apps closed. Averages are taken for all the common smartphone scenarios. Figure 15(a) demonstrates the results. *Spars*-3 reduces the device power consumption by 2.7%, and *Spars*-5 reduces it by 3.0% under the same frame rate with *Sequential*.

6.6 Rendering Scalability

Parallelization allows the OS rendering service to render more graphics primitives under the same time budget (i.e., frame rate), achieving scalability of rendering. *Spars* gives application developers and GUI designers a larger design space for richer page content and more appealing visual effects, especially on large foldable screens, important for the commercial success of smart devices.

We measure the CPU frame rendering time with respect to the number of random graphics primitives (including rounded rectangles, circles, decoded images, text, etc.) that *Spars* and *Sequential* render under the same core configuration and clock rate. Note that this is the lower bound for the number of graphics primitives, as randomness forbids most draw command batching opportunities.

As illustrated in Figure 15(b), for 120Hz frame rate (8.33ms time constraint), *Spars*-3 is able to render 1.62× graphics primitives, and *Spars*-5 is able to render 2.31× primitives compared to the *Sequential* baseline, unleashing more possibilities for the GUI design.

6.7 Costs and Discussions

We discuss the costs of a scalable and parallel rendering service in terms of memory consumption and deployment efforts.

Memory consumption. The primary memory cost arises from the thread creation, with each thread consuming approximately 8MB of memory in the operating system. The additional memory required for the absolute information, self-contained tasks, and overlapping relations does not exceed 10MB. Therefore, for *Spars*-5, the extra memory usage does not exceed 50MB, which is permissible for modern smart devices with ≥8GB of memory.

Deployment efforts. *Spars* is a complete refactoring of the OS rendering service and the 2D drawing engine. A traditional rendering service consists of approximately 200,000 lines of code (LoC) in C++ [24], and a traditional drawing engine contains around 400,000 LoC [28]. To implement a functionally complete scalable and parallel rendering service and drawing engine, we anticipate that modifications required for the refactoring will exceed one third of the source code.

7 Related Work

We present other related work optimizing the rendering services in smart devices in addition to §3.2.

Parallel rendering. Arnau et al. [40] propose parallel frame rendering where consecutive frames can render in parallel to trade responsiveness for energy on mobile GPUs. Skia Graphite backend [85] supports parallel command recording based on tiles (i.e., render passes) in web pages. Other 2D engines [18, 35, 37] enable parallelization for specific tasks, such as image decoding. In comparison, *Spars* can achieve finer-grained parallel rendering necessary for trending foldable and multi-screen devices. Besides, the job systems of game engines [32] support parallelization for physics simulation, AI computation, and rendering preparation in AAA games, while *Spars* is specifically tailored and optimized for operating system 2D rendering.

Scheduling for rendering and displaying. *D-VSync* [87] decouples rendering and displaying to utilize saved computation in short frames for long frames during workload fluctuation. However, *D-VSync* cannot handle scalable and parallel rendering for emerging devices. LTPO [1, 42] supports variable screen refresh rates to save power. Presto [91] reduces touch latency by relaxing synchrony. DSA [45] provides apps a unified view for multiple screens. Other systems [44, 47, 69, 76, 77, 94] implement CPU-GPU power management strategy, e.g., dynamic frequency, to save energy. However, they cannot meet the frame-rate performance demands of foldable and multi-screen devices.

Prediction-based optimizations. Another method to handle the increasing rendering loads is to predict frames in advance. Huang et al. [57], Yan et al. [89], and Baeza-Yates et al. [41] predict the next app to be opened for faster launches. Agrawal et al. [38] reduce activity transition time by pre-inflating UI layouts. PES [49] anticipates DOM events of web apps for better scheduling. Outatime [65] predicts navigation inputs to mask network latency in cloud gaming, while Hou et al. [55] predict head and body motion in VR using LSTM. Although these optimizations function in specific cases, the final effects highly depend on the prediction algorithms and outcomes.

8 Conclusion

This paper presents *Spars* with *Spade*2D, a next-generation scalable and parallel OS rendering service and drawing en-

gine designed for smart devices. The primary innovation lies in its ability to untangle state dependency, drawing order dependency, and interface dependency, allowing out-of-order execution with in-order commit of self-contained rendering tasks. Evaluations demonstrate that *Spars* substantially enhances frame rates on emerging foldable smartphones and multi-screen configurations, efficiently leveraging multiple CPU cores while reducing power consumption and scaling the number of graphics primitives. We anticipate that the design of *Spars* will play a pivotal role in shaping the future development of OS rendering services.

Acknowledgments

We sincerely thank our shepherd and the anonymous OSDI'25 reviewers for their insightful suggestions. This work was supported in part by National Natural Science Foundation of China (No. 62132014, 62432010, 62472279, 62302300), and Startup Fund for Young Faculty at SJTU (SFYF at SJTU). Corresponding authors: Dong Du (dd_nirvana@sjtu.edu.cn) and Haibo Chen (haibochen@sjtu.edu.cn).

References

[1] ipad pro, in 10.5-inch and 12.9-inch models, introduces the world's most advanced display and breakthrough performance, 06 2017. https://www.apple.com/newsroom/2017/06/ipad-pro-10-5-and-12-9-inch-models-introduces-worlds-most-advanced-display-breakthrough-performance/.

[2] Tile-based gpus, 08 2021. https://developer.arm.com/documentation/102662/0100/Tile-based-GPUs.

[3] Graphics subsystem | openharmony docs, 2022. https://gitee.com/openharmony/docs/blob/master/en/readme/graphics.md.

[4] std::shared_mutex - cppreference.com, 06 2023. https://en.cppreference.com/w/cpp/thread/shared_mutex.

[5] Aito m9 smart suv, 2024. https://aito.auto/model/m9/.

[6] Android open source project, 2024. https://source.android.com/.

[7] China passenger car cockpit multi/dual display research report, 2023-2024, 04 2024. http://www.researchinchina.com/Htmls/Report/2024/74971.html.

[8] Directx 12 technology | nvidia, 12 2024. https://www.nvidia.com/en-us/geforce/technologies/dx12/.

[9] Drawing api documentation—harmonyos next, 11 2024. https://developer.huawei.com/consumer/en/doc/harmonyos-references-V5/_drawing-V5.

[10] Flutter - build for any screen, 2024. https://flutter.dev/.

[11] Foldable smartphone market size and share report 2024-2030, 2024. https://www.grandviewresearch.com/industry-analysis/foldable-smartphone-market-report.

[12] The freetype project, 12 2024. https://freetype.org/.

[13] Galaxy z fold6 | unfold the future | samsung us, 12 2024. https://www.samsung.com/us/smartphones/galaxy-z-fold6/1/.

[14] Graphics - android documentation, 08 2024. https://source.android.com/docs/core/graphics.

[15] Huawei mate 70, 2024. https://consumer.huawei.com/cn/phones/mate70/.

[16] Huawei mate x5, 2024. https://consumer.huawei.com/cn/phones/mate-x5/.

[17] Huawei mate xt | ultimate design, 2024. https://consumer.huawei.com/cn/phones/mate-xt-ultimate-design/.

[18] Impeller is a rendering runtime for flutter, 12 2024. https://github.com/flutter/engine/blob/main/impeller/README.md.

[19] ios 18, 2024. https://www.apple.com/ios/ios-18/.

[20] Metal: Accelerate graphics and much more, 12 2024. https://developer.apple.com/metal/.

[21] Multi-core processor market trends, growth report to 2024-2033, 10 2024. https://www.thebusinessresearchcompany.com/report/multi-core-processor-global-market-report.

[22] Openatom openharmony, 2024. https://www.openharmony.cn/mainPlay.

[23] Opengl: The industry's foundation for high performance graphics, 12 2024. https://www.opengl.org/.

[24] Openharmony graphic_graphic_2d, 12 2024. https://gitee.com/openharmony/graphic_graphic_2d.

[25] Oppo find n3 | oppo global, 2024. https://www.oppo.com/en/smartphones/series-find-n/find-n3/.

[26] Power data sources - perfetto tracing docs, 2024. https://perfetto.dev/docs/data-sources/battery-counters.

[27] Skcanvas overview, 2024. https://skia.org/docs/user/api/skcanvas_overview/.

[28] Skia is a complete 2d graphic library for drawing text, geometries, and images., 2024. https://github.com/google/skia.

[29] Skia: The 2d graphics library, 12 2024. https://skia.org/.

[30] Understanding the visionos render pipeline | apple developer documentation, 2024. https://developer.apple.com/documentation/visionos/understanding-the-visionos-render-pipeline.

[31] Understanding user interface responsiveness, 2024. https://developer.apple.com/documentation/xcode/understanding-user-interface-responsiveness/.

[32] Unity - manual: Job system overview, 11 2024. https://docs.unity3d.com/6000.0/Documentation/Manual/job-system-overview.html.

[33] Vsync - android documentation, 08 2024. https://source.android.com/docs/core/graphics/implement-vsync.

[34] Vulkan | cross platform 3d graphics, 2024. https://www.vulkan.org/.

[35] Getting started with qt | qt 6.9, 2025. https://doc.qt.io/qt-6/gettingstarted.html.

[36] Openvg - the standard for vector graphics acceleration, 2025. https://www.khronos.org/openvg/.

[37] Tencent/tgfx: A lightweight 2d graphics library for rendering texts, geometries, and images with high-performance apis that work across various platforms., 04 2025. https://github.com/Tencent/tgfx.

[38] Sumeen Agrawal, Manith Shetty, Sripurna Mutalik, and Anuradha Kanukotla. Method to improve ui rendering using predictive sequence modelling. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 5031–5037, 2022. https://doi.org/10.1109/ICPR56361.2022.9956234.

[39] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007. https://doi.org/10.1109/N-SSC.2007.4785615.

[40] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 83–92, 2013. https://doi.org/10.1109/PACT.2013.6618806.

[41] Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. Predicting the next app that you are going to use. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, page 285–294, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2684822.2685302.

[42] Ting-Kuo Chang, Chin-Wei Lin, and Shihchang Chang. 39-3: Invited paper: Ltpo tft technology for amoleds†. *SID Symposium Digest of Technical Papers*, 50(1):545–548, 2019. https://doi.org/10.1002/sdtp.12978.

[43] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 465–485, Santa Clara, CA, July 2024. USENIX Association. https://www.usenix.org/conference/osdi24/presentation/chen-haibo.

[44] Wei-Ming Chen, Sheng-Wei Cheng, Pi-Cheng Hsiu, and Tei-Wei Kuo. A user-centric cpu-gpu governing framework for 3d games on mobile devices. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 224–231, 2015. https://doi.org/10.1109/ICCAD.2015.7372574.

[45] Zizhan Chen, Siqi Shang, Qihong Wu, Jin Xue, Zhaoyan Shen, and Zili Shao. An old friend is better than two new ones: dual-screen android. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, page 86–98, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3519941.3535071.

[46] Jin Choi, Seohwan Yoo, Hayeon Park, and Chang-Gun Lee. Performance analysis of an embedded chipset on

a multi-screen based automotive applications environment. In *2022 5th International Conference on Information and Computer Technologies (ICICT)*, pages 39–42, 2022. https://doi.org/10.1109/ICICT55905.2022.00015.

[47] Yonghun Choi, Seonghoon Park, and Hojung Cha. Graphics-aware power governing for mobile devices. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, page 469–481, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3307334.3326075.

[48] Christer Ericson. Axis-aligned bounding boxes (aabbs). In *Real-Time Collision Detection*, chapter 4.2, page 77–87. CRC Press, Inc., Boca Raton, FL, USA, 2004. https://doi.org/10.1201/b14581.

[49] Yu Feng and Yuhao Zhu. Pes: Proactive event scheduling for responsive and energy-efficient mobile web computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 66–78, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3307650.3322248.

[50] Francisco Ganacim, Rodolfo S. Lima, Luiz Henrique de Figueiredo, and Diego Nehab. Massively-parallel vector graphics. *ACM Trans. Graph.*, 33(6), November 2014. https://doi.org/10.1145/2661229.2661274.

[51] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. Trinity: High-Performance mobile emulation through graphics projection. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 285–301, Carlsbad, CA, July 2022. USENIX Association. https://www.usenix.org/conference/osdi22/presentation/gao.

[52] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 9–18, New York, NY, USA, 2007. Association for Computing Machinery. https://doi.org/10.1145/1281500.1281665.

[53] Yicheng Gu, Yun Wang, Yunfan Sun, Yuxin Xiang, Xuyan Hu, Zhengwei Qi, and Haibing Guan. gVulkan: Scalable GPU pooling for Pixel-Grained rendering in ray tracing. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1151–1165, Santa Clara, CA, July 2024. USENIX Association. https://www.usenix.org/conference/atc24/presentation/gu-yicheng.

[54] Klaus Hinum. Hisilicon kirin 9010 processor - benchmarks and specs, 07 2024. https://www.notebookcheck.net/HiSilicon-Kirin-9010-Processor-Benchmarks-and-Specs.855471.0.html.

[55] Xueshi Hou and Sujit Dey. Motion prediction and prerendering at the edge to enable ultra-low latency mobile 6dof experiences. *IEEE Open Journal of the Communications Society*, 1:1674–1690, 01 2020. https://doi.org/10.1109/OJCOMS.2020.3032608.

[56] Josh Howarth. Alarming average screen time statistics (2024), 6 2024. https://explodingtopics.com/blog/screen-time-stats.

[57] Ke Huang, Chunhui Zhang, Xiaoxiao Ma, and Guanling Chen. Predicting mobile application usage using contextual information. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, page 1059–1065, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2370216.2370442.

[58] Kristian Høgsberg. Wayland, 2024. https://wayland.freedesktop.org/.

[59] Arseny Kapoulkine. Writing an efficient vulkan renderer, 02 2020. https://zeux.io/2020/02/27/writing-an-efficient-vulkan-renderer/.

[60] Kenny Kerr. Introducing direct2d, 2009. https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/june/introducing-direct2d.

[61] Mark J. Kilgard and Jeff Bolz. Gpu-accelerated path rendering. *ACM Trans. Graph.*, 31(6), November 2012. https://doi.org/10.1145/2366145.2366191.

[62] Youngsok Kim, Jae-Eon Jo, Hanhwi Jang, Minsoo Rhu, Hanjun Kim, and Jangwoo Kim. Gpupd: a fast and scalable multi-gpu architecture using cooperative projection and distribution. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 574–586, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3123939.3123968.

[63] Yoshiyuki Kokojima, Kaoru Sugita, Takahiro Saito, and Takashi Takemoto. Resolution independent rendering of deformable vector objects using graphics hardware. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, page 118–es, New York, NY, USA, 2006. Association for Computing Machinery. https://doi.org/10.1145/1179849.1179997.

[64] Harish Kumar and Anmol Sud. Rendering 2D Vector Graphics on Mobile GPU Devices. In Jirí Bittner and Manuela Waldner, editors, *Eurographics 2021 - Posters*.

The Eurographics Association, 2021. https://doi.org/10.2312/egp.20211025.

[65] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 151–165, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2742647.2742656.

[66] Yang Li, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Feng Qian, Jing Yang, Hao Lin, Yunhao Liu, Bo Xiao, Xiaokang Qin, and Tianyin Xu. Dissecting and streamlining the interactive loop of mobile cloud gaming. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 595–611, Philadelphia, PA, April 2025. USENIX Association. https://www.usenix.org/conference/nsdi25/presentation/li-yang.

[67] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, and Tianyin Xu. Virtual device farms for mobile app testing at scale: A pursuit for fidelity, efficiency, and accessibility. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '23, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3570361.3613259.

[68] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, and Tianyin Xu. Take the blue pill: Pursuing mobile app testing fidelity, efficiency, and accessibility with virtual device farms. *GetMobile: Mobile Comp. and Comm.*, 28(1):5–9, May 2024. https://doi.org/10.1145/3665112.3665114.

[69] Daniel Lo, Taejoon Song, and G. Edward Suh. Prediction-guided performance-energy trade-off for interactive applications. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 508–520, 2015. https://doi.org/10.1145/2830772.2830776.

[70] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph.*, 24(3):1000–1009, July 2005. https://doi.org/10.1145/1073204.1073303.

[71] Bingzheng Ma, Ziqiang Zhang, Yusen Li, Wentong Cai, Gang Wang, and Xiaoguang Liu. Spider: An effective, efficient and robust load scheduler for real-time split frame rendering. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 672–682, 2022. https://doi.org/10.1109/IPDPS53621.2022.00071.

[72] Daan Meysman. Keeping your games 'optimized': Part 1 - triangles, 03 2020. https://www.artstation.com/blogs/daanmeysman/7goy/keeping-your-games-optimized-part-1-triangles.

[73] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[74] Diego Nehab and Hugues Hoppe. Random-access rendering of general vector graphics. *ACM Trans. Graph.*, 27(5), December 2008. https://doi.org/10.1145/1409060.1409088.

[75] Keith Packard and Carl Worth. A realistic 2d drawing system. A rejected SIGGRAPH 2003 submission, 2003. https://keithp.com/~keithp/talks/cairo2003.pdf.

[76] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2744769.2744894.

[77] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2593069.2593151.

[78] Zheng Qin, Michael D. McCool, and Craig S. Kaplan. Real-time texture-mapped vector glyphs. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 125–132, New York, NY, USA, 2006. Association for Computing Machinery. https://doi.org/10.1145/1111411.1111433.

[79] Jiaxing Qiu, Zijie Zhou, Yang Li, Zhenhua Li, Feng Qian, Hao Lin, Di Gao, Haitao Su, Xin Miao, Yunhao Liu, and Tianyin Xu. vsoc: Efficient virtual system-on-chip on heterogeneous hardware. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 558–573, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3694715.3695946.

[80] Xiaowei Ren and Mieszko Lis. Chopin: Scalable graphics rendering in multi-gpu systems via parallel image composition. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*,

pages 709–722, 2021. https://doi.org/10.1109/HPCA51647.2021.00065.

[81] David P. Rodgers. Improvements in multiprocessor system design. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, page 225–231, Washington, DC, USA, 1985. IEEE Computer Society Press. https://doi.org/10.1145/327070.327215.

[82] A.J. Rueda, J. Ruiz de Miras, and F.R. Feito. Gpu-based rendering of curved polygons using simplicial coverings. *Computers & Graphics*, 32(5):581–588, 2008. https://doi.org/10.1016/j.cag.2008.07.005.

[83] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967. https://doi.org/10.1147/rd.111.0025.

[84] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 392–403, New York, NY, USA, 1995. Association for Computing Machinery. https://doi.org/10.1145/223982.224449.

[85] Jim Van Verth. Suggested/status of gpu backends, 11 2024. https://groups.google.com/g/skia-discuss/c/Pd92csb5o4o/m/QlPj80PqAQAJ?utm_medium=email&utm_source=footer.

[86] Steven White, Saisang Cai, Jason Howell, Kent Sharkey, David Coulter, Drew Batchelor, and Michael Satran. Windows gdi - win32 apps, 01 2023. https://learn.microsoft.com/en-us/windows/win32/gdi/windows-gdi.

[87] Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Ming Fu, Binyu Zang, and Haibo Chen. D-vsync: Decoupled rendering and displaying for smartphone graphics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 326–341, New York, NY, USA, 2025. Association for Computing Machinery. https://doi.org/10.1145/3669940.3707235.

[88] Chris Wylie, Gordon Romney, David Evans, and Alan Erdahl. Half-tone perspective drawings by computer. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), page 49–58, New York, NY, USA, 1967. Association for Computing Machinery. https://doi.org/10.1145/1465611.1465619.

[89] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 113–126, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2307636.2307648.

[90] Kinza Yasar. What is web application (web apps) and its benefits, 11 2024. https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app.

[91] Min Hong Yun, Songtao He, and Lin Zhong. Reducing latency by eliminating synchrony. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 331–340, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. https://doi.org/10.1145/3038912.3052557.

[92] Daniel Zhang. How to test whether a car's infotainment system is smooth? 11 car infotainment systems' smoothness comparison, featuring dongchedi's self-developed visual algorithm!, 11 2023. https://www.dongchedi.com/video/7299753097704768015.

[93] Jianwei Zheng, Zhenhua Li, Feng Qian, Wei Liu, Hao Lin, Yunhao Liu, Tianyin Xu, Nan Zhang, Ju Wang, and Cang Zhang. Rethinking process management for interactive mobile systems. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '24, page 215–229, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3636534.3649357.

[94] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149, 2015. https://doi.org/10.1109/HPCA.2015.7056028.



EMT: An OS Framework for New Memory Translation Architectures

Siyuan Chai, Jiyuan Zhang*, Jongyul Kim, Alan Wang, Fan Chung, Jovan Stojkovic,
Weiwei Jia†, Dimitrios Skarlatos‡, Josep Torrellas, Tianyin Xu

University of Illinois Urbana-Champaign †University of Rhode Island ‡Carnegie Mellon University

Abstract

With terabyte-scale memory capacity and memory-intensive workloads, memory translation has become a major performance bottleneck. Many novel hardware schemes are developed to speed up memory translation, but few are experimented with commodity OSes. A main reason is that memory management in major OSes, like Linux, does not have the extensibility to empower emerging hardware schemes.

We develop EMT, a pragmatic framework atop Linux to empower different hardware schemes of memory translation such as radix tree and hash table. EMT provides an architecture-neutral interface that 1) supports diverse memory translation architectures, 2) enables hardware-specific optimizations, 3) accommodates modern hardware and OS complexity, and 4) has negligible overhead over hardwired implementations. We port Linux's memory management onto EMT and show that EMT enables extensibility without sacrificing performance. We use EMT to implement OS support for ECPT and FPT, two recent experimental translation schemes for fast translation; EMT enables us to understand the OS perspective of these architectures and further optimize their designs.

1 Introduction

> "It so happens that a tree format is the only sane format..." [105] —Linus Torvalds, 2002

Virtual memory translation has become a major performance bottleneck of emerging memory-intensive computing [34, 56, 63, 69, 77, 87, 98]. With unprecedented growth of memory capacity, driven by terabyte-scale memory [31, 52] and memory expanders like CXL [71, 96], TLBs cannot scale in the same way as memory. Moreover, emerging workloads like machine learning, graph processing, and bioinformatics have irregular memory access patterns with weak locality, making TLBs and other MMU caches less efficient. As a result, TLB misses are inevitably increasing, resulting in expensive address translation across the memory system.

However, today's translation architectures were designed at a time of scarce memory, and optimize space-efficiency over performance. The *de facto* schemes organize translations into a multi-level radix tree [11, 17, 19]; upon a TLB miss, the MMU must *sequentially* walk the tree, resulting in multiple memory accesses. The x86-64 architecture uses a four-level tree, with a fifth level added in recent hardware [3, 110]. In virtualized environments, translation overhead is magnified by nested translation [39, 79] which takes a two-dimensional walk over the guest and host page tables, resulting in up to 24 sequential memory accesses on four-level page tables.

To address this pressing problem, many new hardware architectures for MMUs have been developed to realize fast translation for today's terabyte-scale, heterogeneous memory. For example, hashing-based translation schemes are revisited [35, 53, 68, 98–100, 109], as hashing is inherently more scalable than walking a tree [109]; a recent hashing scheme, based on Elastic Cuckoo Page Table (ECPT), is reported to reduce translation overhead significantly by enabling parallel lookups of page table entries [98, 100]. New translation schemes using flattened or linear page tables [65, 87] have also been developed. In addition, recent studies advocate for hybrid translation architectures that use different schemes collectively or selectively [26, 27, 49, 57, 63, 111].

Unfortunately, OS support for hardware innovations falls short; few aforementioned new hardware schemes were experimented with commodity OSes like Linux. Instead, evaluations of experimental architectures mostly use performance models to estimate OS overhead [28, 65, 77, 95, 109, 111], or trace-driven simulation with traces collected by running workloads on unmodified Linux [77, 95, 98, 109]. The assumption is that OS overhead on different translation architectures is constant. However, our paper shows that translation architectures could have significant impacts on OS performance.

In fact, the difficulties of OS support has affected hardware research—disruptive hardware designs are often considered "*undesirable*" and lose to incremental approaches (see [77]). Our discussions with hardware vendors tell us that the lack of commodity OS support and evaluation is a major barrier to assessing and adopting new hardware translation schemes.

The unsatisfactory OS effort is largely due to memory management systems in commodity OSes not having the extensibility for emerging translation architectures. For example, Linux assumes a radix-tree based page table structure and lacks extensibility to support hardware schemes that cannot fit in its tree definition. As a result, supporting a new architecture often requires heavy modifications of memory management code in architecture-independent modules. Essentially, Linux provides no extensible interface for memory translation, un-

*Co-lead of the EMT project

like its other subsystems (e.g., VFS for file systems [51]).

Contributions. We develop a pragmatic OS framework and toolchains atop Linux to embrace new hardware translation architectures for today's memory technologies. We term our framework Extensible Memory Translation (EMT). We target Linux as it is still the *de facto* commodity OS and is mostly assumed by architecture research on memory systems.

EMT provides an architecture-neutral Linux interface that 1) supports diverse memory translation architectures, 2) enables hardware-specific optimizations, 3) accommodates modern hardware and OS complexity, and 4) has negligible overhead over Linux's hardwired implementation. By abstracting translation-related operations, EMT simplifies the effort to support and experiment with new translation architectures on Linux. With EMT, developers only need to implement an MMU driver as a hardware-specific implementation of translation logic, without modifying the architecture-independent memory management code. EMT also makes it easy to profile and analyze OS performance with regard to hardware translation schemes as it abstracts translation-related operations.

We implement EMT on Linux (v5.15), referred to EMT-Linux. We modularize architecture-independent code with EMT API, removing hardwired assumptions. EMT realizes negligible overhead through careful engineering and optimizations (with less than 0.5% overhead on average across key OS operations). Moreover, EMT-Linux realizes all existing features and hardware-specific optimizations in vanilla Linux.

We build on EMT-Linux to add OS support for ECPT [98] and FPT [87], two new translation schemes. Porting these new hardware schemes on Linux without a framework like EMT would require major rewriting of Linux's memory management module. With EMT, supporting them on Linux is modularized with manageable engineering efforts.

Evaluating new hardware schemes on EMT-Linux faces a common challenge of hardware-software codesign—the hardware is not yet available to run the OS. To address this problem, we assemble a toolchain that runs EMT-Linux on QEMU with an emulated MMU where we implement the hardware translation logic. Our toolchain enables us to understand the OS perspective of new translation architectures, such as its OS overhead over x86. The toolchain also supports cycle-accurate hardware simulation.

We share our experience of supporting new translation schemes on EMT-Linux, which enables us to understand OS memory management challenges beyond hardware perspectives. We present our reflection on the ECPT design and address correctness challenges such as managing the kernel page table (kECPT) and the paradox involved in changing the translations of the kECPT itself and of the kernel code that manages kECPT, as well as performance challenges like efficient locking and memory scanning. Arguably, OS framework support is essential to encourage and embrace disruptive hardware innovations, and EMT is an important step forward.

Summary. This paper makes the following contributions:

- A discussion on empowering new, experimental hardware-assisted translation architectures on commodity OSes.
- EMT as an extensible framework for developing and evaluating OS memory management on new translation architectures, and its implementation on Linux.
- An experience of building ECPT and FPT on Linux using EMT and the reflection on hardware/OS designs.
- An open platform for developing, testing, and evaluating OS kernels on new memory translation architectures.
- Our artifacts: https://github.com/xlab-uiuc/emt.

2 Background

2.1 Memory Translation Hardware

Modern computer systems use hardware-assisted memory translation, where the translation schemes are defined by the hardware architecture. Upon a TLB miss, the MMU searches the translation structures (e.g., a page table) to obtain the translation—the virtual-to-physical address mapping.

x86 Translation Scheme. All x86 processors since Intel 80386 have used a radix tree, as depicted in Figure 1. The depth of this tree has increased from two levels in 80386 to four levels in x86-64, with the fifth level upcoming [3, 110] and already supported in Linux [43].

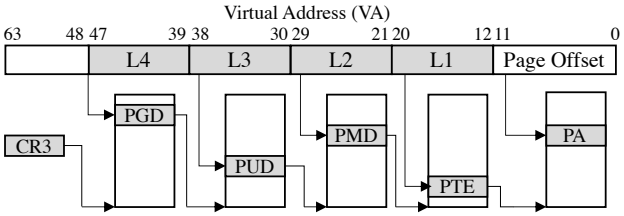


Figure 1: **Radix-tree based page table walk in x86-64 ISA.**

The main advantage of radix-tree based translation is space efficiency—tables at each level are created on demand—if at any level, no page is allocated within an address range, the sub-tree is not allocated. This yields significant memory savings, as the address space of typical applications is sparse.

However, tree-based translation suffers from a major performance drawback—it must *sequentially* walk the tree with multiple memory accesses. Page walk caches [32, 40, 87] and huge pages [69, 81] are used to reduce the length of the walks, but struggle to address emerging workloads with large memory footprints and weak-locality access patterns. The overhead is further magnified in *nested translation* for virtualized environments. The MMU performs a two-dimensional walk over the guest and the host page tables. A nested translation takes up to 24 sequential memory accesses with four-level page tables, and up to 35 with five-level tables. It is reported that nested translation can take more than 50% of the execution time of memory-intensive workloads [27, 48, 77].

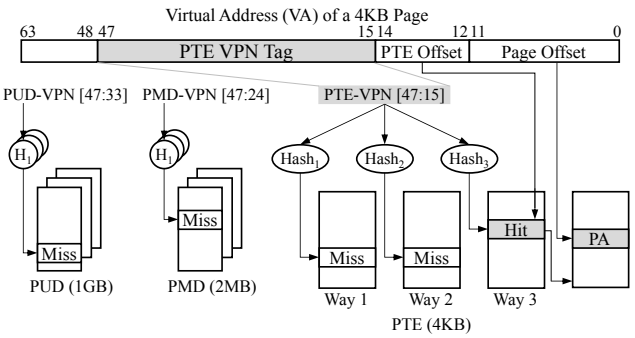


Figure 2: **Parallel page-table lookup in an ECPT-based architecture [98] using three-way cuckoo hashing.**

```
1  void vunmap_pmd_range(pmd, addr, end) {
2    pmd_t *pmd = pmd_offset(pud, addr);
3    do { ...
4      // try remove huge page entry
5      int cleared = pmd_clear_huge(pmd);
6      ...
7      if (pmd_none_or_clear_bad(pmd))
8        continue;
9
10     // try remove underlying PTEs
11     vunmap_pte_range(pmd, addr, next, mask);
12     ...
13   }
14   while (pmd++, addr = next, addr != end);
15 } /* mm/vmalloc.c */
```

Find a 2MB entry (PMD) from the addr. of a 1GB entry (PUD).

Overloaded semantics:
(1) The entry does not point to a 2MB huge page;
(2) The entry is not a directory;
(3) No entries of 4KB pages in the 2MB address range.

The iterator assumes the spatial locality of entries.

Figure 3: **Examples of Linux memory management code that is hardwired to radix-tree based translation.**

Flattened Page Table (FPT). Recently, extensive efforts are being made to optimize memory translation architectures. A common principle is to shorten page table walks of the x86-64 scheme [26, 28, 48, 65, 77, 87, 111]. A recent proposal from Arm, known as Flattened Page Table or FPT [87, 106], flattens the x86 page tables by dynamically merging intermediate tree levels to reduce indirections and prioritize caching of page table entries. Specifically, FPT tries to merge L4 and L3, as well as L2 and L1, in Figure 1, shortening the walk by half.

Hashing-based Translation and ECPT. Hashing is being actively revisited for translation [35, 53, 63, 68, 98–100, 109]. We focus on ECPT (Elastic Cuckoo Page Table) [98], a new hashing-based scheme which effectively speeds up translation with fully parallel lookups of page table entries.

Figure 2 depicts ECPT-based memory translation. Different from conventional hashed page tables [46, 54, 60, 104, 109], ECPT uses process-private hashed page tables that are dynamically resized based on occupancy. ECPT uses Cuckoo Hashing [83] and maintains multiple tables (called *ways*) to proactively resolve hash collisions by moving entries across ways. During a page table walks, the MMU computes hashes of the Virtual Page Number (VPN) to look up the ways in parallel. It eliminates the need for sequential tree walks. ECPT supports multiple page sizes (e.g., 1GB, 2MB, and 4KB pages) by maintaining a set of hash page tables for each page size.

To reduce parallel lookups, ECPT uses an in-memory data structure named Cuckoo Walk Table (CWT); each CWT maintains metadata (size and way) of ECPT translation entries. CWTs are cached in special MMU caches named Cuckoo Walk Caches (CWCs). During translation, if a requested page hits a CWC, the MMU can directly look up the specific page table (with the size) or the specific way.

2.2 OS Memory Management

The OS manages in-memory translation structures like page tables and other auxiliary structures (e.g., CWT in ECPT), defined by the hardware architecture. The OS is also responsible for managing translation data (virtual-to-physical address mappings and metadata like protection and dirty bits).

Translation information is consumed by almost all OS memory management operations. Hence, translation architectures have strong implications on OS performance. For example, recent work [108] shows that the majority of page migration cost is from OS memory management on translation-related operations like unmapping/remapping pages and demoting huge pages, instead of actual page copies.

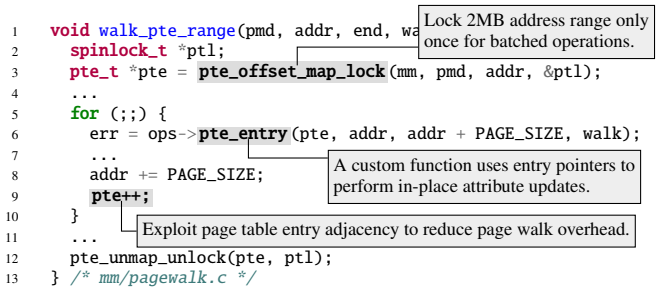
Modern OSes separate *architecture-independent* OS memory management and *architecture-dependent* hardware support. Mach [25] designed a machine-independent memory manager where its architecture-independent code makes few assumptions about MMUs [91]; the design is inherited by BSD kernels [45]. EMT is inspired by Mach/BSD, but focuses on empowering new, experimental translation schemes.

Linux layers machine-independent/dependent code differently. It maintains a multi-level tree-based page table in the architecture-independent module. This design achieves high-performance memory management: 1) it enables optimizations that need to directly manipulate page tables and translation entries, and 2) it avoids overhead due to indirections. However, it lacks extensibility to different translation architectures especially those that do not fit its tree definition such as ECPT or even the conventional hashed page tables.[1]

3 The Need for a New OS Framework

Emerging memory technologies and research on MMU architectures pose a strong need of developing and evaluating OS memory management on new translation schemes. The current practice of evaluating new translation schemes mostly relies on hardware simulation, either using performance models to estimate OS overhead [28, 65, 77, 95, 109, 111], or replaying traces collected by running workloads on vanilla Linux [77, 95, 98, 109]. Such approaches can hardly capture complex OS-architecture interactions [29]. We argue that the lack of OS effort is largely due to memory management systems in commodity OSes not providing an extensible interface for new, different translation architectures.

[1] Hashed Page Tables (HPTs) was provided by architectures like IA-64 and POWER [1, 10], but Linux does not have native support for HPTs [105]. Linux maintains a software tree-based page table to store full virtual memory mapping and treats the HPTs as soft-managed extended TLBs. Such an approach leads to duplication of translation data and redundant maintenance.




```
1  void walk_pte_range(pmd, addr, end, wa
2    spinlock_t *ptl;
3    pte_t *pte = pte_offset_map_lock(mm, pmd, addr, &ptl);
4    ...
5    for (;;) {
6      err = ops->pte_entry(pte, addr, addr + PAGE_SIZE, walk);
7      ...
8      addr += PAGE_SIZE;
9      pte++;
10   }
11   ...
12   pte_unmap_unlock(pte, ptl);
13 } /* mm/pagewalk.c */
```

Figure 4: **Hardware-specific optimizations in Linux.**

Hardwiring Translation Schemes Is Untenable. Linux's memory management code is currently hardwired to a five-level tree. Without an extensible framework, it can hardly embrace new translation architectures that use different lookup structures such as hash tables [98, 99, 109], flattened or range tables [26, 65, 87], and hybrid schemes [27, 63].

Figure 3 shows a representative example in Linux, where a memory operation for unmapping a virtual address range at PMD level (`vunmap_pmd_range` in `vmalloc.c`) makes the following assumptions specific to Linux's tree definition:

- A translation entry of a 2MB virtual address range (PMD) can be found from the entry of a 1GB address range.
- An entry of a 2MB address range (PMD) either points to a 2MB huge page or a directory of 4KB pages.
- The translation entry of the next 2MB address range can be iterated by increasing the pointer.

Such implementation patterns are commonplace in Linux's current architecture-independent memory management module. Hence, supporting a different translation architecture would require heavy rewriting of existing Linux code.

In fact, it is also nontrivial for Linux to support translation schemes that use a different tree. Evidently, adding the fifth level took 715 lines of code changes across 23 files [97], all in the architecture-independent modules of Linux. Similarly, though FPT's design strives to limit OS changes [87], it still introduces nontrivial changes of Linux's architecture-independent code to fold the intermediate levels of the tree.

Furthermore, without a well-defined interface, memory management code becomes hard to maintain due to implicit assumptions and semantic overloads with continuous optimizations and fixes. In Figure 3, `pmd_none_or_clear_bad()` overloads three semantics, making the code hard to maintain.

Hardware-Specific Optimizations Are Desired. Inspired by Mach/BSD [13, 45, 91] that have clean separation of machine-independent and dependent code, we started from a pmap-like interface for Linux. In Mach/BSD, architecture-indepedent memory management code uses pmap interface to manage virtual memory mappings. pmap [91] represents a virtual-to-physical address map, which can point to a VAX linear page table [70] or segment registers in IBM RT PC. With pmap, different translation schemes can be supported by writing pmap modules. Typical pmap operations include creation, deletion, and update of address mappings [13, 91].

Unfortunately, we find that a map interface like pmap is not sufficiently expressive to write hardware-specific optimizations, especially in the Linux context. For example, pmap routines for inserting or finding a translation, only reference a virtual address without exposing translation entries. Figure 4 shows three common patterns of optimizations in existing Linux code: 1) batching with fine-grained locks for range operations, 2) in-place translation entry updates with no data copy, and 3) direct fetching of translation entries by offsets. None of them are easy to write using a map, as they need to directly manipulate translation entries. As optimizations of memory management are critical to Linux performance—27.8% of Linux kernel patches to memory management are for performance optimizations [59]—a simple map interface makes it hard to fully empower translation architectures.

Existing Frameworks Do Not Target Translation. Driven by new memory technologies like tiered memory, several new memory management frameworks [61, 92, 103] have been developed to improve the extensibility of Linux's memory manager. However, few of them concern hardware translation architectures, but focus on application-specific page prefetching and replacement policies. FBMM [103] proposes to reuse VFS [51] and write memory managers as file systems. It exposes the `page_fault` VFS callback for managing page table entries. However, FBMM relies on Linux's existing code that is hardwired to the multi-level radix tree structure, and thus cannot support different hardware translation schemes. Fundamentally, file system interfaces can hardly serve as an effective framework for memory translation.

4 EMT Design

Extensible Memory Translation (EMT) is an OS framework built atop Linux with the goal of empowering new, experimental memory translation architectures. Figure 5 gives a high-level overview of EMT. EMT currently focuses on supporting new memory translation architectures in the OS kernel, instead of exposing them to userspace [47]. The design and implementation of EMT achieves the following goals:

- **EMT is an architecture-neutral framework.** EMT is not hardwired to specific translation structures such as multi-level tree in Linux. Supporting a new memory translation scheme should not change architecture-independent code.
- **EMT enables hardware-specific optimizations.** EMT allows architecture-dependent code to customize routines for hardware-specific optimizations. The ability to customize differs EMT from high-level interfaces like pmap in Mach/BSD [91] and hat in SunOS [50].
- **EMT is modularized and improves maintainability.** EMT provides well-defined translation semantics and organizes them with an object-oriented design. The interface

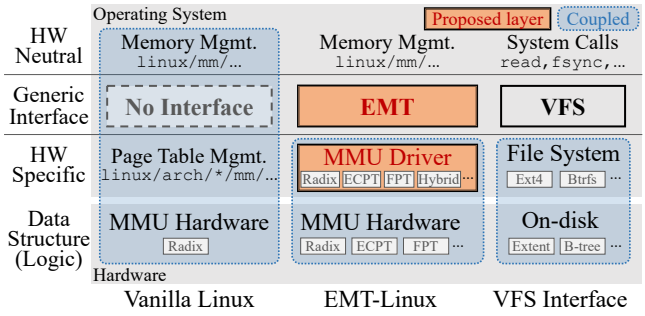


Figure 5: **Overview of the EMT framework, compared with vanilla Linux and analogized to VFS.**

eliminates architecture-specific or overloaded semantics. EMT also makes it easy to profile and analyze OS performance with regard to translation architectures.

- **EMT has negligible overhead.** EMT is carefully implemented with compiler optimization, cache efficiency, and the ability to inline functions. EMT-Linux serves as a performance baseline for new architectures or OS modules.
- **EMT accommodates modern hardware and OS complexity.** EMT aims to develop insights into complex OS-architecture interactions. EMT supports *all* Linux's memory management features related to CPU MMU translation such as huge pages, swapping, DAX memory, etc.

4.1 EMT API

EMT exposes an object-oriented API that organizes translation related kernel routines around three architecture-neutral primitives: 1) *translation object* that maintains all the information of a virtual-to-physical address translation, 2) *translation database* that stores translation objects of an address space, and 3) *translation service* that manages the MMU states.

In EMT, all architecture-dependent code is implemented in drivers for MMU architectures, referred to as *MMU drivers*. To support a new translation scheme on EMT-Linux, one needs to write the MMU driver that implements EMT API without the need to modify architecture-independent code.

A key design principle of the EMT API is to abstract the *functions* of translation for OS memory management from hardware-defined *shapes*. Despite many different translation architectures (§2), the essential function of memory translation is to output the virtual-to-physical address mapping and all associated metadata, which is abstracted as a *translation object* in EMT. EMT avoids assuming shapes of translation data like translation entry schema or translation data structures. How to encode translation information of hardware-defined shapes into the EMT API is the job of MMU drivers.

The EMT API is organized as 1) *basic functions* that must be implemented by each MMU driver for its supporting translation architecture, and 2) *customizable functions* that have a default implementation that is architecture-independent; the default implementation only uses basic functions and other

```
//// Translation Object
// Read the attribute of an object based on the attribute key
tobj_read_attr(tobj, attr_key) -> (error, attr_value)
// Update the value of a given attribute in an object.
tobj_write_attr(tobj, attr_key, new_val) -> (error, old_val)
...
//// Translation Database
// Find a translation object in a translation database
tdb_find_tobj(tdb, vaddr) -> (error, tobj)
// Update a translation object matching tobj's va
// by copying attributes of tobj.
tdb_update_tobj(tdb, tobj) -> (error, old_tobj)
// Remove the given translation object from the database.
tdb_remove_tobj(tdb, tobj) -> (error, old_tobj)
...
//// Translation Service
// Switch the translation database of the current address space
tsvc_switch_tdb(tdb) -> (error, out_tdb)
// Read the translation database of a given address space
tsvc_read_tdb(cpu) -> (error, out_tdb)
...
```

Figure 6: **Examples of basic functions in the EMT API.** EMT exposes 15 basic functions in total.

common code. These routines can be customized by MMU drivers for hardware-specific optimizations.

The basic functions represent the minimal set of functions to support a new translation architecture. Developers can start with basic functions to achieve a functional kernel and then iteratively add optimizations via customizable functions. Basic functions free developers from sophisticated optimizations in Linux; it leads to a more friendly development and test cycle.

4.1.1 Basic Functions

Basic functions are expected to be architecture-dependent and are required to be implemented by each MMU driver. Figure 6 gives examples of basic function APIs. The API design avoids assuming specific hardware schemes like the five-level radix tree (§2) or page table entry schema. It also prevents overloaded semantics like `pmd_none_or_clear_bad` (Figure 3) due to software evolution.

Translation Object. A translation object encodes a virtual-to-physical address mapping and all its associated metadata. For paged architectures, it encodes translation information stored in page table entries. For tree-based architectures, a translation object encodes information in page table entries that point to the physical page (see §5). A translation object can also represent segments or variable-length memory regions. Translation metadata are encoded as attributes such as size, protection, presence, swap, etc., which are queried by general OS memory management. EMT lets MMU drivers encode metadata into architecture-dependent bits. MMU drivers can also encode architecture-specific attributes and use them to implement architecture-specific features like protection keys and capabilities [19, 21, 107].

Translation Database. Translation objects of an address space are stored in a conceptual database. The database is commonly implemented by a page table (which can be of various shapes) in an MMU driver. It can also be implemented by multiple co-existing page tables (e.g., ECPT needs separate

```
//// Translation Object Iterator
// Return the current translation object and advance the iterator
tobj_iter_next(iter) -> (error, tobj)
...
//// Huge Page
// Check if a given virtual address range can be a huge page
thp_eligible(tobj, pg_size, vma) -> eligible
...
//// Address Range
// Check if the given virtual address range has no mapping in it
addr_range_void(tdb, start, end) -> is_void
...
//// Lock
// Get a lock to protect all translation objects in the range
addr_range_get_lock(tdb, start, end) -> (error, tlock)
...
//// Swap
// Get the Linux swp_entry_t from a translation object
tobj_to_swap(tdb, tobj) -> (error, swp_entry)
...
```

Figure 7: **Example customizable functions in the EMT API.** EMT exposes 35 customizable functions in total in 7 groups.

page tables for each userspace process and for a shared kernel space across processes; see §6), or by segments or VMA registers [63, 111]. EMT requires a translation database to return one and only one translation object for a virtual address, as the function of the database. EMT does not concern the shape of the database. Its basic functions abstract away architecture-specific structures.

The EMT API decouples translation objects from translation databases; the former does not concern how translation data are stored, while the latter does not interpret translation objects. The benefit is modularity, making it easy to reuse existing code (as shown by the FPT and ECPT MMU drivers).

Translation Service. A translation service abstracts the MMU of a system. It is the root of the EMT API, and manages the creation, destruction, and switching of address spaces. Upon a context switch, the translation service is called to switch the databases. Translation service decouples translation management from task management.

4.1.2 Customizable Functions

Customizable functions provide an interface for MMU drivers to implement hardware-specific optimizations. Each customizable function has a default architecture-neutral implementation using basic functions and other architecture-independent code. Figure 7 gives examples of customizable function APIs.

Customizable functions are exposed to MMU drivers via a combination of redefinable macros, following Linux's convention. The architecture-neutral version is wrapped in `#ifndef` so that it can be used if no architecture-specific implementation is available. If an MMU driver wants to implement a customizable function, it defines the customizable function name to its own implementation; otherwise, the `#ifndef` redirects the interface to the default implementation. Since customizable functions always have an architecture-neutral implementation, adding new customizable functions in EMT (if needed in the future) will not break existing MMU drivers.

In principle, customizable functions are those that can benefit from architecture-specific optimizations. The choice now is through high-level reasoning together with profiling that identifies performance-critical code. Currently, EMT exposes different groups of customizable functions, as exemplified by Figure 7. We discuss the iterator group as an example.

```
1  int tobj_iter_next(struct tobj_iter *iter, struct tobj *tobj)
2  { ...
3    int ret = tdb_find_tobj(iter->tdb, iter->va, tobj);
4    ret = tobj_read_attr(tobj, TOBJ_ATTR_SIZE, &size);
5    if (!ret) iter->va += size;
6    return ret;
7  } /* mm/emt-generic.c */
```

(a) Default (architecture neutral)

```
1  int tobj_iter_next(struct tobj_iter *iter, struct tobj *tobj)
2  { ...
3    // handling the most common case of iterating in a 2MB range
4    if (iter->ptep) {
5      tobj->va = iter->va;
6      tobj->pte = iter->pte;
7      if ((iter->va + PAGE_SIZE) & (~PMD_MASK)) {
8        iter->va += PAGE_SIZE;
9        iter->pte++; // Exploit radix's spatial locality
10       return 0;
11     }
12     // Cross 2MB boundary, update ptep based on pmd
13   }
14   ... // handling other cases.
15 } /* arch/x86/mm/radix.c */
```

(b) x86-64 radix MMU driver

Figure 8: **A customizable function of the iterator.**

Translation-Object Iterator. Iterating over a large number of translation objects is a common management pattern when the OS scans a memory region (e.g., for page migration [14] and huge-page promotion [16]). The performance of such operations is critical, e.g., an optimized iterator can reduce page fault handling cost by 52.5% (§8.4). EMT provides a translation-object iterator with customizable functions. Figure 8a shows the default implementation of the iterator's `tobj_iter_next()` function, which is not efficient—the OS needs to walk from the root to the leaves of the page-table tree in every iteration using `tdb_find_tobj`. Note that such OS walks cannot benefit from hardware caches like PWCs as they are done by software. Figure 8b shows the customized implementation of the x86-64 MMU driver that leverages spatial locality of the radix tree to directly increment the pointer to get the next object. We discuss the implementation of the ECPT MMU driver in §6.2.

EMT's iterator functions differ from Linux's page-table iterator [90]—it does not assume the radix tree structure. EMT's iterator does not return intermediate entries (which are specific to tree schemes), but returns translation objects.

4.2 Generality

It is hard to prove generality, but our effort on supporting x86-64 radix tree, FPT, and ECPT indicates that EMT can express different hardware schemes, and *all* related optimizations. These schemes represent tree- and hashing-based translation designs, which ground many emerging architectures. A few new architectures [27, 56, 63, 111] propose hybrid designs which often use a fast scheme for common patterns and a

slow scheme for correctness [56, 63, 111], or expose multiple schemes to userspace [27, 47]. For pure hardware-based fast schemes [56, 111], the translation is still the traditional one from an OS perspective (e.g., Midgard [56] uses radix tree as its backend). For architectures that use multiple schemes for different address space segments, the MMU drivers can manage multiple page tables under the hood of a database.

From a metadata perspective, EMT can support address translation schemes with coupled or separate metadata management. For metadata that are separately stored from translation structures, the MMU driver encodes and updates the translation entry and its metadata separately in the translation object. For example, EMT supports protection keys like Intel MPK [21] and hardware capabilities [107] by encoding keys as attributes of translation objects. EMT naturally supports capability-based translation [42, 107] which encodes permissions into capabilities (protected pointers).

EMT also supports mechanisms that manage cache modes like PAT [74] and MTRR [72]. EMT views the cache modes of a page as attributes of its translation object. When the cache mode of a page is changed, the MMU driver updates the attributes. For example, the x86 MMU driver will update the PAT, PCD, and PWT bits of its translation entry [82].

Limitations. EMT only concerns translation, and does not change operations on physical memory pages. EMT focuses CPU virtual-to-physical translation and does not target IOMMU translation for DMA requests. Linux code for IOMMU translation is also hardwired to the radix-tree scheme; the EMT approach can potentially apply. EMT assumes that programs run with virtual addresses; EMT currently does not support designs that make program use physical addresses directly [101, 102].

EMT currently only supports the native environment; virtualization support is on the roadmap. As the extended page table [19] (EPT) has a similar structure as the native page table, we plan to create a virtualization-specific MMU driver to manage EPT. The driver creates a translation database for each VM and handles tasks like changing EPTP registers upon VM switches and allocating EPT tables. KVM can call EMT APIs accordingly. We expect paravirtualization support (`pv_ops.mmu` [73, 75] in particular) to be encapsulated in the MMU driver: MMU drivers with paravirtualization support can call `pv_ops.mmu` for function patching.

5 EMT-Linux with x86-64 MMU Drivers

We develop EMT on Linux, referred to as EMT-Linux. Conceptually, it took four steps: 1) identifying memory management code in architecture-independent modules that are hardwired to x86-like, tree-based translation scheme; 2) rewriting them using the EMT API; 3) moving architecture-specific optimizations into the x86-64 MMU driver, and 4) writing default implementations of customizable functions.

The EMT implementation on Linux (v5.15) takes 9.5K

```
1  void walk_tobj_range(start, end, walk) {
2    struct tlock lock; struct tobj_iter iter;
3    struct tobj tobj; ulong size;
4    struct tdb *tdb = walk->mm->tdb;
5    err = addr_range_get_lock(tdb, start, end, &lock);
6    addr_range_write_lock(&lock);
7    err = tobj_iter_init(tdb, start, end, &iter);
8    ...
9    while (tobj_iter_has_next(iter)) {
10     err = tobj_iter_next(&iter, &tobj);
11     err = tobj_read_attr(&tobj, TOBJ_ATTR_SIZE, &size);
12     err = walk->ops->tobj(tobj, tobj->va, tobj->va + size,
       ↪ walk);
13     ...
14   }
15   tobj_iter_end(&iter);
16   addr_range_write_unlock(&lock);
17   addr_range_put_lock(&lock);
18 } /* mm/pagewalk.c */
```

Customizable split PT lock.

Customizable iterator can be implemented as ++ for performance.

EMT also allows in-place operations via low-level abstractions.

Figure 9: **Rewriting code in Figure 4 with EMT.**

lines of code changes (7.3K for interface refactoring and 2.2K for the x86-64 MMU driver) in 15 person-months. We changed 196 kernel functions in the `mm` directory of Linux—most memory management code interacts with the page table.

We use macros and inline functions to minimize interface overhead as per Linux's coding style [20]; `#ifndef`-based function redirections are used otherwise. Specifically, we turn functions no more than three lines into macro or inline functions. We ensure that EMT does not break performance characteristics, e.g., no increase of stack size or call stacks, and no decrease of instruction cache hit rates in most cases (§8.3).

Moreover, we preserve *all* of Linux's existing architecture-specific optimizations and realize them in the x86-64 MMU driver. Hence, EMT supports all virtual memory features of vanilla Linux and is *transparent* to user applications.

OS Memory Management with EMT. In EMT-Linux, OS memory management is no longer hardwired to specific translation schemes. Meanwhile, EMT-Linux reserves hardware-specific optimizations like in Linux—both basic and customizable functions can be instantiated by different MMU drivers. Figure 9 shows the code in EMT-Linux that implements the hardwired Linux routine in Figure 4. It uses the translation-object iterator (see Figure 8) to scan all the pages in a memory region. This is a common pattern used in many OS memory management operations such as page promotion and migration, as well as page eviction (e.g., using LRU and MGLRU).

Figure 10 shows a skeleton of the page fault handler in EMT-Linux. In vanilla Linux, the page fault handler walks down the page-table tree level by level to determine the type of faults and dispatch them to corresponding subroutines (e.g., a leaf-level PMD entry implies a 2MB page). Differently, EMT-Linux's page fault handler decouples page sizes from the translation data structures (the radix tree). It looks at the translation object and the corresponding page size based on the faulting address, and decides the subroutine to invoke.

Swapping operates on translation objects that map pages to be swapped. Different architectures may encode swap-page information differently. x86-64 MMU driver stores swap in-

```
static vm_fault_t __handle_mm_fault(vma, vaddr, flags)
{ ...
  struct tobj tobj;
  struct tdb *tdb = vma->vm_mm->tdb;
  err = tdb_find_tobj(tdb, vaddr, &tobj);
  err = tobj_read_attr(&tobj, TOBJ_ATTR_MAPPED, &mapped);
  ...
  if (mapped) {
    if (flags & FAULT_FLAG_WRITE) { // fix the write fault
      err = tobj_read_attr(&tobj, TOBJ_ATTR_WRITE, &write);
      if (!write)
        return do_wp_page(&vmf);
    }
    ... // many other fixes
  } else { // mapping does not exist;
    ...
    while (pg_size > BASE_PAGE_SIZE) { // try huge page
      if (thp_eligible(tobj, pg_size, vma)) {
        int ret = create_huge_page(&vmf, pg_size);
        ...
      }
      pg_size = dec_page_size(tdb, pg_size);
    }
    if(vma_is_anonymous(vma)) { // handle anon page fault
      struct page *page = alloc_page_vma(...);
      if (!page) goto oom;
      struct tobj old_tobj;
      err = tobj_update_attr(&tobj, TOBJ_ATTR_PA,
            page_to_pfn(page) << BASE_PAGE_SHIFT);
      ... // update other attributes like permissions
      err = addr_range_get_lock(tdb, vaddr,
            vaddr + BASE_PAGE_SIZE, &lock);
      addr_range_write_lock(&lock); // lock the range
      err = tdb_update_tobj(tdb, &tobj, &old_tobj);
      addr_range_write_unlock(&lock);
      addr_range_put_lock(&lock);
      return 0;
    }
    ... // handle other type of base page faults
  }
  ... // error handling (e.g., oom)
} /* mm/memory.c */
```

Figure 10: **Snippet of EMT-Linux's page fault handler.**

formation in a page table entry if the present bit is cleared; huge page swapping [44] is done by encoding PMD entries. ECPT and FPT use the same mechanism. EMT provides customizable functions if architectures need different encoding.

EMT encapsulates translation cache management in MMU drivers, as the cache structure is architecture-specific. For example, ECPT does not use x86-64 PWCs but has CWCs to cache page size and way information [98]. TLB-related code is handled with the same principle. The MMU driver performs TLB flushes and guarantees the consistency of cache state. Batch invalidation is supported by customizable functions.

x86-64 MMU Driver. For the x86-64 radix-tree translation scheme, our x86-64 MMU driver maintains the multi-level tree page table. The MMU driver encodes page table entries at all levels of the radix tree for a given virtual-to-physical address mapping in the translation object. This encoding enables the MMU driver to realize all the optimizations specialized for the tree-based x86-64 page table, while making architecture-neutral code agnostic. For example, the MMU driver can directly operate on the PMD or PUD entries to implement Linux's split page table lock [15] through the `addr_range_get_lock` customizable function (Figure 9). The MMU driver can also implement optimizations that leverage spatial locality of the tree structure, as exemplified by the iterator (Figure 8). Encoding attributes of x86 page table entries via translation object API is straightforward by reading/writing bits in the entries, and our x86-64 MMU driver supports hardware-specific features like Intel MPK.

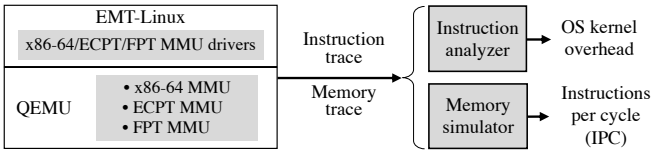


Figure 11: **Emulator toolchain for OS development and evaluation on experimental memory architectures.**

FPT MMU Driver. We implemented an MMU driver for a new translation scheme based on Flattened Page Table (FPT) [87]. The FPT design aims to minimize OS changes [106] and is based on the x86-64 radix-tree page table. Without an interface like EMT, it needs nontrivial changes to Linux's architecture-independent code, including changing macros that define bits in page table entries and checking if a level needs to be folded. The Linux prototype in the FPT paper [87] supports flattening of L3+L2 only. We wrote the FPT MMU driver by reusing the x86-64 MMU driver code with 664 lines of C code. With the EMT API, no architecture-neutral OS code needs to be changed. Our FPT MMU driver supports all three types of flattening patterns of tree levels, and it co-exists with the x86-64 radix MMU driver.

6 ECPT on EMT-Linux

6.1 Emulator-based Toolchain

It is challenging to develop OSes for new translation architectures without manufactured MMU hardware. We did not find an available toolchain for developing and evaluating OS kernels with experimental architectures like ECPT. The original evaluation of ECPT [98] collects memory traces from simulation using Simics [76] on *vanilla Linux* and replays traces in SST [94] that simulated an ECPT MMU. Simics is closed-source and only supports existing ISAs.

We develop an emulator-based toolchain using QEMU [5] (Figure 11). We use QEMU's software MMU mechanism [4] to develop an emulated ECPT MMU where we implement ECPT translation logic and hardware caches in 3.1K lines of C code. QEMU offers an emulated x86-64 MMU which serves as a baseline for performance evaluation.

Our toolchain can connect to trace-driven hardware simulators (QEMU provides no cycle-accurate simulation). We developed instruction and memory tracers using QEMU's TCG plugin [23]. The instruction trace helps analyze kernel and userspace behavior (§8.4), as well as hardware simulation.

6.2 ECPT MMU Driver

We implement an ECPT MMU driver on EMT based on the hardware architecture design [98]. We use a three-way

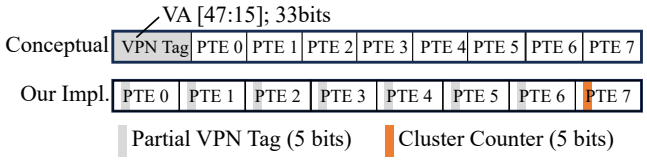


Figure 12: **Implementation of 64-byte PTE-ECPT entry clusters consists of eight 8-byte PTE entries.** Each entry contributes five bits for the VPN tag or a cluster count.

ECPT for each page size (4KB/2MB/1GB) so the translation database manages nine control registers, one per way; registers are updated upon a process context switch. The driver also manages nine registers for the kernel page tables (§7.1). The ECPT MMU driver takes 7.4K lines of C code.

The ECPT structures are implemented in the ECPT MMU driver. For memory efficiency, we allocate ECPTs of a specific size on demand. We follow the ECPT design of clustering eight translation entries into a 64-byte cache line to improve locality. To do so, we construct the VPN tag (Figure 2) with bits from multiple translation entries, as shown in Figure 12. For the PTE VPN tag, each entry contributes five bits so the first seven entries together make up the VPN tag, and the last entry contributes five bits as the count of valid entries in the cluster. The count is used to calculate occupancy (for table resizing) and for freeing the cluster. These metadata are managed by basic functions. PUD and PMD entries use the same design but with 24 bits and 15 bits for the VPN tag.

When a table's occupancy exceeds a threshold (0.6 as in [98]), a background kernel thread resizes the table and migrates entries from the old table to the new table gradually.

Except for the repurposed five bits for metadata, ECPT's translation entry by design follows x86-64 PTE format. We reuse the PTE encoding from the x86 radix MMU driver, enjoying the benefit of EMT's object-oriented design.

Cuckoo Walk Tables (CWTs) are implemented in the ECPT MMU driver, invisible to architecture-independent code. We follow the ECPT design [98] to use a 5-bit section header (a section is the address range translated by one ECPT translation entry) and cluster 64 headers into a 64-byte cache line. Each header is one byte and contributes three bits to encode the VPN tag and count of valid section headers.

Optimizations. We implemented a series of optimizations through the customizable functions, driven by benchmarking and profiling (§8.4). For example, we customize the iterator with architecture-specific optimizations to accelerate range operations. The idea is to exploit locality within a translation entry cluster: the iterator finds the next entry by pointer arithmetic, instead of hashing-based lookups, when the entry is not the last one in its cluster. One common optimization pattern is to minimize the cost of range operations—the default implementation (e.g., Figure 8a) often takes too many fine-grained hashing-based operations.

7 Reflection on ECPT Design

We show that building OS components is essential to understanding architecture designs. The efforts on developing and evaluating ECPT with EMT-Linux reveals OS challenges of using a fast translation scheme, which were unexpected (undocumented in the original hardware design).

7.1 Managing Kernel Page Tables

In modern OSes like Linux, the kernel address space is shared across processes [9]. In tree-based translation, sharing is realized by having high-level page table entries pointing to the shared subtree of the kernel address space [22]. This design does not apply to hashing-based translation. One option is to maintain one ECPT for each process containing both user- and kernel-space addresses. However, such a design is memory inefficient and leads to high overhead, e.g., once a translation of a kernel-space address is updated, the OS must update the corresponding entries in ECPTs of all processes.

Our ECPT MMU driver maintains a global kernel-space ECPT (kECPT) shared among processes and an independent user-space ECPT (uECPT) for every process. The kECPT has the same page size and way configuration as each uECPT. The kECPT and uECPTs are managed independently. When KPTI (kernel page table isolation) [22] is enabled, two independent global kECPTs are managed (a complete kECPT and a minimal kECPT). This design requires ECPT MMUs to expose another nine control registers that point to kECPT(s).[2]

Self-Reference Paradox. Managing kECPTs is more challenging than uECPTs. Different from uECPTs that are mandated by the kernel, the kECPT manages the kernel's own code and data—translations for kECPT management code and kECPT itself are stored in kECPT. This creates a challenge for ECPT which needs to move translation entries across ways for resolving hash collisions—in the moving window, if the entries map the kECPT itself or the kernel code for moving entries are missing, the kernel crashes due to missing translation of the kECPT or code, as shown in Figure 13. Frequent triggers are kernel's huge page promotion/demotion which remove entries and insert new entries in different tables.

We term the issue *self-reference paradox*. The root cause is a lack of hardware support for atomic updates on multiple memory locations. Radix-tree page tables do not face this paradox, because they do not move entries and the huge-page promotion/demotion is done by updating one PUD/PMD entry. In general, the self-reference paradox can happen in kernel page tables that need to move entries (many advanced index schemes need to move entries [36, 113]). Note that

[2] The cost of additional architectural registers is negligible; they add to less than 0.01% of the die area. The area overhead was measured with the same methodology as in [98, 100] using CACTI [30] with 22nm technology. The new design also distinguishes whether the target address belongs to the upper or lower half of the address space and directs the lookups to uECPT or kECPT. CWC is shared and has no additional cost.

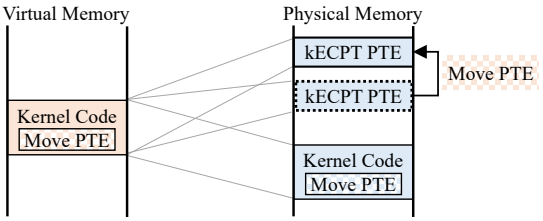


Figure 13: **An example of self-reference paradox caused by the need of moving kECPT entries that are needed to find kernel code for moving kECPT entries (§7.1).**

other kernel structures like forwarding table [112] do not face this paradox if they do not map kernel code/data.

We address the paradox via a series of software-hardware endeavors. First, we always copy the entry to the new location *before* removing it at the old location, which may cause duplicated entries but no missing entries. We instruct the MMU to handle duplicate entries that have the same virtual-to-physical address mapping and the same protection bits; the deduplication policy also works for huge-page promotion/demotion as it does not change the mapping or protection. We avoid location-changing read-update-write operations in OS code so that duplicated entries always have the same mapping and protection. For ECPT, any operation that may change the content of an entry can only be done after the entry is locked.

Atomic kECPT Switching. Another form of self-reference paradox manifests via KPTI [22]—when switching between the kernel space and the user space, the OS needs to switch between a full kECPT and a minimal kECPT. One potential implementation is to use multiple `mov` instructions to redirect control registers from the old kECPT ways to the new ways. However, this may lead to an inconsistent ECPT state during the switch window, where some registers point to new ways and the others point to old ways. This leads to issues when the translations of currently executing kernel code are stored in different ways across the new and old kECPTs. Note that simply keeping context switch code covered by both kECPT is not enough since the instructions can reside in different ways of the old and new tables. Instructions might not be successfully fetched when kECPT state is inconsistent.

Solving this problem needs a mechanism for atomic switching from all old kECPT ways to the new kECPT ways. Such an atomic switching mechanism can be realized with hardware support. Our solution is to add additional sets of kECPT control registers and the hardware switches between the two sets of control registers, in a similar vein as the `VMLAUNCH` and `VMRESUME` instructions of x86 on updating multiple registers together. The switch is a serializing instruction to ensure that all instructions after it will use the new kECPT.

7.2 Implications on OS Performance

Managing Sparse Address Space. A fundamental efficiency property of tree-based page tables is the ability to manage sparse address space, which is critical for OS memory management which commonly needs to scan large, but sparse address ranges. In tree page tables, a high-level entry can encode properties of a large address range, e.g., a nonexistent PMD entry indicates no valid page allocated in the corresponding 2MB range. Hashed page tables have no such hierarchical relationship between entries; hence, the OS may go through all the possible entries in the large, sparse address range.

One optimization is to enable a similar property in ECPT by designing special entries that are not for translation, but for encoding states. The OS checks the entry to learn the states of the corresponding address range, e.g., the OS can query such an entry to check if the corresponding 2MB address range has any allocated page, instead of checking all possible PTE entries. The design would need to cooperate with the MMU.

Multicore Scalability. We find it nontrivial to implement efficient locking primitives similar to Linux's split page table lock [15] in ECPT, because the location of an ECPT entry can be moved due to insertion or rehashing. The movement of entries makes it prone to deadlocks, e.g., one thread t holding a lock l and inserting a new entry, thus consequently requires moving an existing entry e (see [83]) which is locked by another thread that waiting on l. One solution is to let t release all its acquired locks and retry later when deadlock. The complexity lies in recording and rewinding states changed by t. We are exploring an alternative design that introduces a separate lock table to provide the OS with the flexibility of implementing lock primitives, which has the semantic of locking both an address range and the related entries. Our implementation of the ECPT MMU driver has no correctness issue, as we use a coarse-grained lock.

8 Evaluation

8.1 Methodology

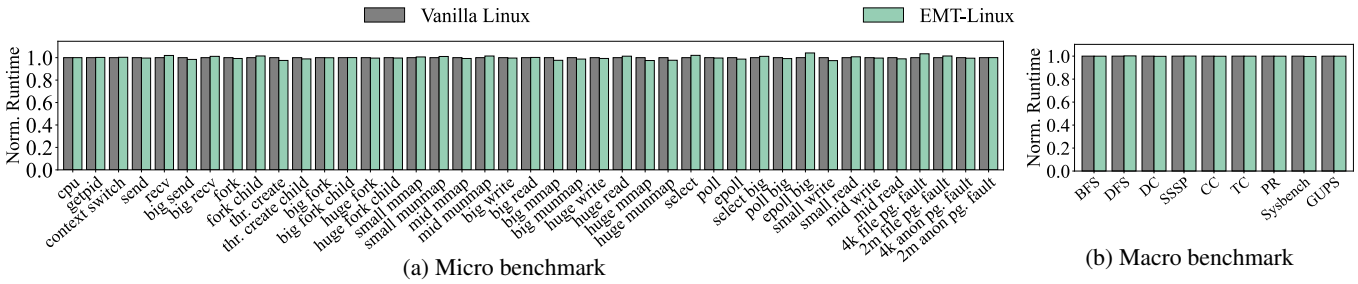
We validate the correctness of the EMT-Linux implementation using Linux tests. Since EMT supports all virtual memory features in Linux, EMT-Linux should pass any userspace tests.

We also measure the interface overhead of EMT by running the same set of performance benchmarks on EMT-Linux (§5) and vanilla Linux running upon the same hardware.

We then measure the performance of EMT-Linux on ECPT. We run EMT-Linux with the ECPT MMU driver on top of an emulated ECPT MMU using our emulation framework (§6.1). We compare the performance of EMT-Linux with the Radix MMU driver running on an emulated x86-64 MMU.

Benchmarks. We use LEBench [93] as a micro benchmark to measure the performance of core OS operations such as page fault handling, context switching, and system calls.

We use nine memory-intensive macro benchmarks that stress the TLB and need off-TLB translation. We use the same set of macro benchmarks from the ECPT paper [98, 100] except two where we failed to reproduce the working set.



(a) Micro benchmark (b) Macro benchmark

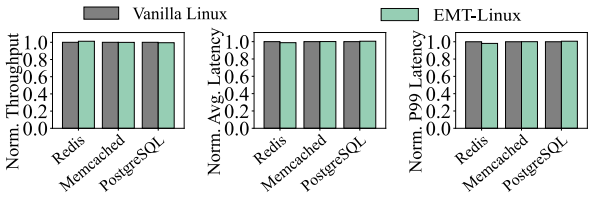
Figure 14: **Performance overhead of EMT-Linux over vanilla Linux (measured by the micro and macro benchmarks).**

The macro benchmarks include seven applications from the GraphBIG benchmark suite [80]: Breadth First Search (BFS), Depth First Search (DFS), Degree Centrality (DC), Single Source Shortest Path (SSSP), Connected Components (CC), Triangle Count (TC), and PageRank (PR); they use the LDBC-1000K dataset [2], with working sets of about 8.5 GB. We also run the GUPS benchmark [58] which issues random memory updates, and a memory test from Sysbench [66]. Both GUPS and Sysbench have 64 GB working sets.

We also evaluate three memory-intensive applications: Redis, Memcached, and PostgreSQL. Table 1 shows the workloads. All these applications are multithreaded programs.

Application	Working Set	# Records	Read:Write	# Requests
Redis	128 GB	536 M	50:50	60 M
Memcached	69 GB	56 M	80:20	10 M
PostgreSQL	64 GB	21 M	100:0	25 M

Table 1: **Application workloads used in the evaluation.**



(a) Throughput (b) Avg. Latency (c) P99 Latency

Figure 15: **Performance overhead of EMT-Linux over vanilla Linux on real-world applications**

8.2 Functional Correctness

We show that EMT supports all memory management features of Linux by running Linux Test Project (LTP) [12] against EMT-Linux. We use the default kernel configuration for the generic kernel (5.15.0-125-generic) of Ubuntu 20.04. LTP includes 1,405 tests in total and 1,208 of them apply to the kernel configuration. EMT-Linux with Radix, ECPT, and FPT MMU drivers both pass all 1,208 tests which covers 376 system calls. We also cross-validated program outputs of the micro and macro benchmarks across EMT-Linux and vanilla Linux. We use testing as a continuous effort throughout our development, rather than a one-time effort, which helped capture bugs in a timely manner.

8.3 EMT Interface Overhead

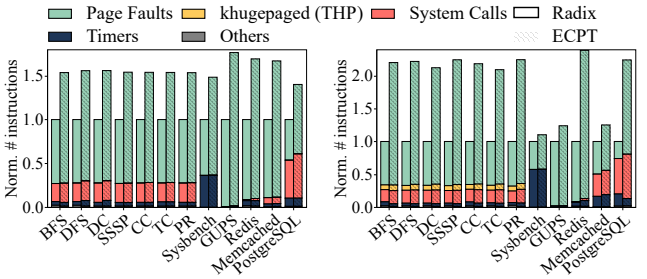
EMT introduces negligible overhead. We run the benchmarks and application workloads on vanilla Linux and EMT-Linux running on the same hardware. The hardware is a dual-socket Intel Xeon Gold 6346 server at 3.10GHz with 16 cores and 256GB DDR4-3200 DRAM. We disable hyperthreading and fix core frequency to make the measurement stable. The overhead of EMT is calculated by normalizing the results of EMT-Linux to vanilla Linux. We experimented with various kernel configurations (e.g., enabling THP); the results are consistent.

Figure 14a shows the results of kernel micro benchmarks LEBench [93]. EMT-Linux exhibits an average normalized kernel-routine performance of 99.9% relative to vanilla Linux (with a standard deviation of 1.1%) across the 41 LEBench microbenchmarks. The largest overhead comes from the "epoll big" benchmark, where EMT-Linux slows the benchmark by 4.2%. It was caused by not inlining certain functions due to coding style restriction, which can be further optimized by restructuring the code or forcing inlining. Figure 14b shows that the overhead of EMT-Linux is less than 0.1% on the macro benchmarks. Figure 15 shows the normalized throughput, average latency, and tail latency of three real-world applications (with workloads in Table 1) on EMT-Linux, normalized to their performance on vanilla Linux. The measured differences of the three metrics are within 0.1%, 0.1%, and 0.2%, respectively. The results of the real-world application show that the EMT interface does not affect system responsiveness.

The negligible overhead is attributed to two kinds of endeavors. First, we carefully engineered the EMT interface to preserve the performance characteristics—we minimize increased stack size or deepened call stacks, and maintain cache efficiency. Instruction cache hit rates on EMT-Linux differ from those on vanilla Linux by at most 0.8%. Second, EMT enables us to implement all the hardware-specific optimizations in the MMU drivers.

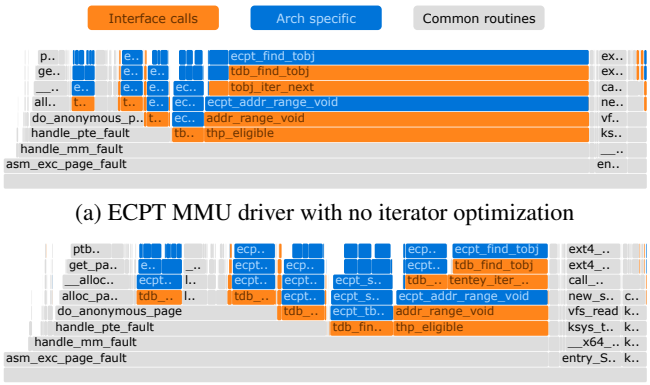
8.4 OS Performance on ECPT

We show that EMT enables us to understand OS performance on new translation architectures using ECPT as an example. We evaluate ECPT by running EMT-Linux with the ECPT MMU driver on our emulation framework and comparing



(a) 4KB (b) THP

Figure 16: **Distribution of kernel instructions of EMT-Linux with the Radix and ECPT MMU drivers.**



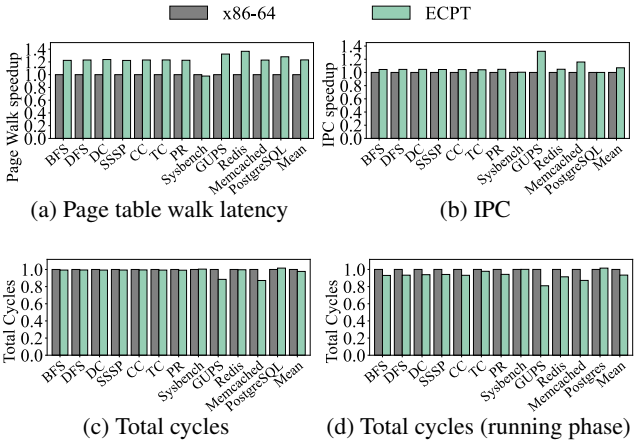
(a) ECPT MMU driver with no iterator optimization

(b) ECPT MMU driver with iterator optimization

Figure 17: **Flame graph of EMT-Linux kernel instructions when running GraphBig BFS (THP enabled).**

its performance with the x86-64 radix MMU driver. We run macro benchmarks and application workloads and record all kernel and user instructions (with both 4KB pages and THP).

Understanding OS Overhead of ECPT. Figure 16 shows the distribution of kernel instructions; most of them are for page fault handling. Compared with Radix, ECPT spends 1.74x more instructions on page fault handling on average for 4KB pages, and 2.59x more with THP enabled. When THP is enabled, ECPT leads to relatively more work than Radix because Linux's THP implementation uses a few expensive operations to check if there exist valid entries in a given address range. These checks are on the critical path of page fault handling when THP is enabled since the kernel needs to know if the 2MB range has any mappings before it can decide if a 2MB huge page mapping should be built. For Radix, these checks are cheap (0.39% of total kernel work) because a PMD entry has information on PTE tables. If the entry is not valid or present, then neither 2MB nor 4KB mappings exist; the huge-page bit tells if the entry points to a huge page or a directory of 4KB pages. However, these checks are expensive in ECPT, as ECPT's entries are independent, the kernel may need to check all 512 4KB entries of pages in the 2MB range, which requires many expensive lookup operations. We discussed the potential solution in §7.2.



(a) Page table walk latency (b) IPC

(c) Total cycles (d) Total cycles (running phase)

Figure 18: **ECPT hardware simulation results— Page Table Walk Latency, Instruction Per Cycle (IPC) and Total Cycles—with the ECPT system and the x86-64 system, running EMT-Linux.** All results are normalized to those of the x86-64 system as the baselines.

Effectiveness of Optimizations. EMT exposes optimization opportunities for MMU drivers using customizable functions. Figure 17 uses flame graphs to show the effectiveness of iterator optimization running GraphBIG BFS with THP. With the iterator optimization (§6.2), ECPT can save 49.0% of total kernel work, and 52.5% of the page fault handling work. The optimization drastically saves the work of hashing and lookups of the architecture-neutral default implementation, by incrementing the pointer to an entry directly, as long as the entry is within an entry cluster (Figure 12). Note that such kernel work as software overhead cannot benefit from hardware caches such as TLBs or PWCs.

8.5 Hardware Simulation

8.5.1 ECPT

EMT's emulator toolchain provides an open platform of hardware simulation for experimental architectures where silicon implementations are not available. To demonstrate this, we run hardware simulations for EMT-Linux with the x86-64 radix and ECPT MMU drivers using the DynamoRIO simulator [41]. We use the hardware configuration described in the ECPT paper [98]. For PWC configuration, we follow more recent works [77, 111] to use 3 levels of PWC with 2-4-32 entries per level. We use the macro benchmarks and the three application workloads described in §8.1.

Traditional Hardware Metrics. Our toolchain measures traditional hardware metrics, namely Page Table Walk Latency and Instructions Per Cycle (IPC), which are considered key performance metrics of hardware translation architectures (see [87, 98]). As shown in Figures 18a and 18b, ECPT has significant performance advantages over x86-64 (Radix). On average, ECPT speeds up page table walks by 23.1%, and

increases IPC by 7.0%. For most benchmarks, ECPT shows nontrivial page table walk and IPC speedups, as it directly accesses the last-level PTEs. The only exception is Sysbench whose workload has a high PWC hit rate of 99.8% at the PMD level on x86-64, i.e., the x86-64 page table walks skip most intermediate steps and directly accesses last-level PTEs; meanwhile, ECPT pays extra cycles for hash computation in addition to CWC accesses, so it slightly slows down Sysbench's page table walk by 2.2%

New Metrics with OS Overhead. We find that IPC, which measures instruction throughput, does not fully reflect application runtime—though ECPT increases IPC, our ECPT MMU driver introduces more kernel instructions than the x86-64 MMU driver (§8.4). Hence, we measure the total cycles for running macro benchmarks and applications, including the OS overhead (mostly page-fault handling as shown in Figure 16). Figure 18c shows that, on average, the ECPT system reduces total cycles by 2.3% across the workloads. GUPS and Memcached show major benefits, where the ECPT system reduces the total cycles by 11.5% and 12.9%, because page table walks account for over 66% of their execution time.

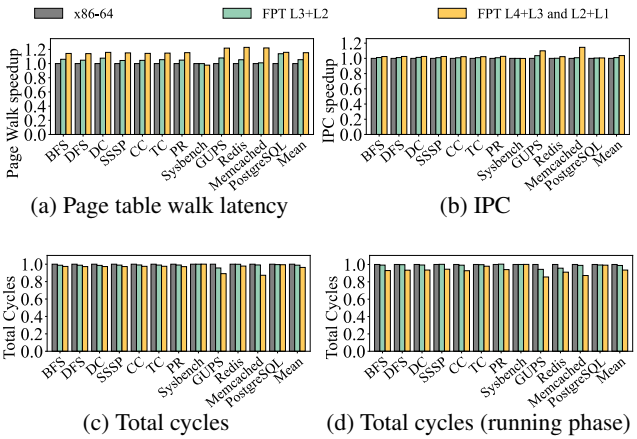
The aforementioned analysis shows that the OS can play an important role in application performance, which could be overlooked with hardware metrics. Without building the OS memory management, it is hard to evaluate OS overhead accurately. Prior work assumed that OS overhead is constant under different memory-translation architectures. In the initial modeling, the memory trace is collected from running benchmarks on vanilla Linux on x86-64; the trace is then replayed in a simulator of ECPT MMU. As shown in §8.4, memory translation architectures can have profound implications for OS performance. EMT aims to enable OS experience for new translation architectures. We believe that EMT effectively lowers the barrier to implement OS support for new memory-translation architectures by making the kernel developers focus on MMU driver implementations.

IPC as a metric aligns with application's execution time when OS overhead is minimal. All the evaluated workloads have two phases: a *loading phase* that loads data from the disk to memory, and a *running phase* that runs the computation or serves user requests. The running phase involves less kernel work with kernel instructions accounting for less than 1.3% of the total instructions. With only the running phase considered, ECPT improves IPC by 7.5% and reduces total cycles by 6.6% compared to the x86-64 system (Figure 18d).

The results could be different with different hardware simulators. We expect the impact of kernel instructions to become smaller when using SST [94] which models OoO execution more precisely. The SST evaluation remains future work.

8.5.2 FPT

We start with a FPT configuration that flattens L3+L2 tables (see Figure 1) which was implemented by the original OS prototype [87]. Our evaluation shows that the benefit of flattening L3+L2 is limited: only L3 entries were saved, which were cached effectively in x86-64. As shown in Figures 19a, 19b, 19c, compared with Linux on x86-64 (baseline), on average, the FPT-based system speeds up page table walk latency by 5.5%, IPC by 1.0%, and the total cycles by 1.1%.



(a) Page table walk latency (b) IPC

(c) Total cycles (d) Total cycles (running phase)

Figure 19: **Hardware simulation results of the FPT system and the x86-64 system, running EMT-Linux.** All results are normalized to those of the x86-64 system as the baselines.

We then implemented the FPT configuration that flattens both L4+L3 and L2+L1. This configuration was not implemented in [87] due to its difficulties of supporting 2MB huge pages. We implemented a 4KB-only version and evaluated it with only 4KB base pages (no huge page). With this configuration, compared with Linux on x86-64, the FPT-based system speeds up page table walk latency by 15.3%, IPC by 3.6%, and total cycles by 3.7% (Figures 19a, 19b, and 19c).

Flattening both L4+L3 and L2+L1 in FPT yields performance comparable to ECPT in the running phase. Compared with radix, ECPT and FPT reduce total cycles by 6.6% and 6.4%, respectively (Figures 18d and 19d). For hardware metrics (page table walk latency and IPC), the FPT-based system matched ECPT-based system's performance: ECPT is only faster than FPT by 2.4% in page table walk latency and 0.4% in IPC. On the other hand, ECPT inherently supports huge pages of varying sizes, unlike this FPT configuration.

9 Experience and Lessons Learned

Developing EMT-Linux and the MMU drivers together with the emulated MMU took extensive engineering efforts. Specifically, we had to develop interacting moving parts across the hardware-software boundary.

A key principle is to enable incremental, contiguous engineering practice. For example, when implementing ECPT, our first milestone is to write a basic per-process hashed page table (BHPT) using EMT, without features like collision resolution, elastic resizing, CWT/CWC, etc. We implement BHPT logic in both EMT-Linux and the MMU and have a running

system that only supports a tiny microbenchmark. Despite being basic, the running system serves as a foundation for gradually adding features and eventually evolving into a full-fledged system. For features that need both hardware and software support, we start from the hardware, which typically has simpler logic and mostly reads translation data.

We continuously test and evaluate our system, not only for correctness (§8.2) but also for performance. Continuous profiling helped us quickly observe performance regression over vanilla Linux; when developing ECPT, profiling helped us identify major inefficiencies compared to the radix-page-table-based system. We developed the instruction analysis with flame graphs [55] in our emulator toolchain. and use them extensively. We also added GDB support for the ECPT-based system (QEMU's GDB was hardcoded with radix).

We largely reuse the existing Linux compiler toolchain to keep the effort manageable. We leverage x86's model-specific registers (MSR) [19] as the control registers for ECPT, which is supported by GCC (x86's read/write MSR instruction allows access to arbitrary MSR with a 32-bit identifier). We use Clang's static analysis tools (e.g., clang-query [18]) to search for error-prone code patterns during refactoring.

One mistake we made was to start from Linux's boot-time kernel page table [6, 8], which is legacy code without debugging tools. Developing boot-time kECPT requires implementation in assembly and build mappings of kernel code/data from a bootstrap address space. We spent four months to understand the details of the boot-time kernel page table. Retrospectively, we should have started from runtime kECPTs (by switching the system to it after boot [7]) so we could have a running system quicker to parallelize our efforts.

10 Related Work

As memory translation has become a major bottleneck of emerging memory-hungry, irregular workloads such as generative models, graph analytics, and recommendation systems, translation architecture has been an immensely active research topic recently. Extensive efforts are made on new designs of TLB [33, 86, 88, 89] and page walk caches [32, 40], and more efficient huge page management [62, 69, 81, 84, 85].

To fundamentally resolve translation bottlenecks, recent work [53, 56, 63, 98, 109] is actively rethinking translation architectures; a common thread is to prioritize speed over space efficiency (with abundant memory) and use fast lookup structures to reduce tail latency (avoiding pointer chasing).

Memory translation architectures have profound implications on OS reliability and performance, but unfortunately have not been well explored in prior work. EMT is designed to enable development and evaluation of OS support for emerging translation architectures to formulate an OS perspective.

Note that memory management and its extensibility were well studied in OS research [24, 38, 47, 67, 91] and are continuously revisited, driven by the growth of memory capacity and the heterogeneity introduced by memory tiering [37, 61, 71, 78, 92, 103]. So far, extensibility refers to the ability for user space to customize memory management features, e.g., page fault handler. Few prior studies considered extensibility in terms of empowering memory translation architectures or studied their implications on OS performance.

We are inspired by machine-independent memory management designs in Mach [91] and SunOS [50]. The goal of EMT is to enable the OS development for emerging, experimental memory translation architectures and demonstrate the value with our experience. We attempted to simplify the EMT interface, e.g., to a map interface [50, 91]; however, we find it hard to balance architecture neutrality and low-level optimizations. One way is to add hints, but hints are not expressive for passing information to different MMU drivers.

Virtuoso [64] is a recent hardware simulation framework that considers OS overhead (mostly the overhead of page fault handling) using an imitated MimicOS in the user space. EMT and Virtuoso are fundamentally different but are complementary. Unlike Virtuoso which is designed for architecture-centric simulations, EMT is an OS interface for enabling OS developement and evaluation with regard to new translation architectures. EMT benefits architecture designs from an OS perspective by running real MMU drivers on Linux.

11 Concluding Remarks

EMT is an OS framework for developing and evaluating OS memory management for new memory translation architectures. Our work shows the importance of understanding hardware translation schemes from the OS perspective. Specifically, we show that fast translation schemes like ECPT can incur new challenges for the OS. We will release and make EMT and the emulator toolchains as an open platform and encourage new hardware architectures to experiment with modern OSes. With the significant diversity of emerging workloads and increasing heterogeneity of interconnected memory devices, it becomes harder to foresee a one-size-fits-all translation scheme. Hence, OS extensibility for different translation schemes is critical to enable specialized translation. We hope that EMT design starts a practical journey towards extensible OS kernels for translation of heterogeneous devices.

Acknowledgement

We thank the anonymous reviewers, and our shepherd, Haggai Eran, for their insightful comments. We thank Chloe Alverti, Ren Wang, Andrea Arcangeli, Jinghao Jia, and Kaiyang Zhao for their feedback and discussion. We are very grateful to Shanbo Zhang and Rubin Du who helped prepare the EMT artifacts and earn the Reproducibility badge. We also thank Divyam Arora, Logan Cudia, and Yuchen Tang for their participation. This work was funded in part by NSF CNS-1956007, CNS-2145295, CNS-2239311, CNS-2107307, an IBM-Illinois Discovery Accelerator Institute (IIDAI) grant, an Intel TSA grant, and an Intel SRS Award.

References

[1] Intel® IA-64 Architecture Software Developer's Manual. http://refspecs.linux-foundation.org/IA64-softdevman-vol2.pdf, July 2000.

[2] GraphBIG Dataset. https://github.com/graphbig/graphBIG/wiki/GraphBIG-Dataset, Mar. 2015.

[3] 5-Level Paging and 5-Level EPT White Paper. https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html, May 2017.

[4] Features/SoftMMU. https://wiki.qemu.org/Features/SoftMMU, Jan. 2018.

[5] QEMU: A generic and open source machine emulator and virtualizer. http://www.qemu.org, June 2020.

[6] arch/x86/boot/compressed/head_64.S. https://github.com/torvalds/linux/blob/v5.15/arch/x86/boot/compressed/head_64.S, Aug. 2021.

[7] arch/x86/kernel/cpu/common.c. https://github.com/torvalds/linux/blob/v5.15/arch/x86/kernel/cpu/common.c, Oct. 2021.

[8] arch/x86/kernel/head64.c. https://github.com/torvalds/linux/blob/v5.15/arch/x86/kernel/head64.c, May 2021.

[9] Memory Management. https://www.kernel.org/doc/html/v5.15/x86/x86_64/mm.html, Aug. 2021.

[10] Power ISA™ Version 3.1B. https://openpowerfoundation.org/specifications/isa/, Sept. 2021.

[11] AMD64 Architecture Programmer's Manual. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf, June 2023.

[12] Linux Test Project. https://github.com/linux-test-project/ltp/tree/20230929, Sept. 2023.

[13] Machine address mapping definitions – machine-independent section. https://github.com/freebsd/freebsd-src/blob/release/14.0.0/sys/vm/pmap.h, Aug. 2023.

[14] Page migration. https://www.kernel.org/doc/html/next/mm/page_migration.html, Aug. 2023.

[15] Split page table lock. https://www.kernel.org/doc/html/next/mm/split_page_table_lock.html, Aug. 2023.

[16] Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html, Dec. 2023.

[17] Arm® Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/, Mar. 2024.

[18] clang-query tool. https://github.com/llvm/llvm-project/tree/main/clang-tools-extra/clang-query, Jan. 2024.

[19] Intel® 64 and IA-32 Architectures Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, Mar. 2024.

[20] Linux kernel coding style - The Linux Kernel documentation. https://www.kernel.org/doc/html/v6.12/process/coding-style.html#the-inline-disease, Sept. 2024.

[21] Memory Protection Keys. https://docs.kernel.org/core-api/protection-keys.html, Oct. 2024.

[22] Page Table Isolation (PTI). https://www.kernel.org/doc/html/next/x86/pti.html, Jan. 2024.

[23] QEMU TCG Plugins. https://www.qemu.org/docs/master/devel/tcg-plugins.html, Feb. 2024.

[24] ABROSSIMOV, E., ROZIER, M., AND SHAPIRO, M. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)* (Nov. 1989).

[25] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 Summer USENIX Technical Conference (USENIX Summer '86)* (July 1986).

[26] AHN, J., JIN, S., AND HUH, J. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA-39)* (June 2012).

[27] ALAM, H., ZHANG, T., EREZ, M., AND ETSION, Y. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA-44)* (June 2017).

[28] ALVERTI, C., PSOMADAKIS, S., KARAKOSTAS, V., GANDHI, J., NIKAS, K., GOUMAS, G., AND KOZIRIS, N. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA-47)* (June 2020).

[29] ANDERSON, T. E., LEVY, H. M., BERSHAD, B. N., AND LAZOWSKA, E. D. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)* (Apr. 1991).

[30] BALASUBRAMONIAN, R., KAHNG, A. B., MURALIMANOHAR, N., SHAFIEE, A., AND SRINIVAS, V. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).

[31] BARR, J. EC2 High Memory Update - New 18 TB and 24 TB Instances. https://aws.amazon.com/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/, Oct. 2019.

[32] BARR, T. W., COX, A. L., AND RIXNER, S. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37)* (June 2010).

[33] BARR, T. W., COX, A. L., AND RIXNER, S. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA-38)* (Sept. 2011).

[34] BASU, A., GANDHI, J., CHANG, J., HILL, M. D., AND SWIFT, M. M. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA-40)* (June 2013).

[35] BENDER, M. A., BHATTACHARJEE, A., CONWAY, A., FARACH-COLTON, M., JOHNSON, R., KANNAN, S., KUSZMAUL, W., MUKHERJEE, N., PORTER, D., TAGLIAVINI, G., VOROBYEVA, J., AND WEST, E. Paging and the Address-Translation Problem. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21)* (July 2021).

[36] BENDER, M. A., CONWAY, A., FARACH-COLTON, M., KUSZMAUL, W., AND TAGLIAVINI, G. Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once. *Journal of the ACM 70*, 6 (Nov. 2023), 1–55.

[37] BERGMAN, S., FALDU, P., GROT, B., VILANOVA, L., AND SILBERSTEIN, M. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management (ISMM '22)* (June 2022).

[38] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Dec. 1995).

[39] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)* (Mar. 2008).

[40] BHATTACHARJEE, A. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)* (Dec. 2013).

[41] BRUENING, D. L. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Sept. 2004.

[42] CARTER, N. P., KECKLER, S. W., AND DALLY, W. J. Hardware Support for Fast Capability-based Addressing.

[43] CORBET, J. Five-level page tables. https://lwn.net/Articles/717293/, Mar. 2017.

[44] CORBET, J. The final step for huge-page swapping. https://lwn.net/Articles/758677/, July 2018.

[45] CRANOR, C. D., AND PARULKAR, G. M. The UVM Virtual Memory System. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX '99)* (June 1999).

[46] DOUGAN, C., MACKERRAS, P., AND YODAIKEN, V. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)* (Feb. 1999).

[47] ENGLER, D. R., GUPTA, S. K., AND KAASHOEK, M. F. AVM: Application-Level Virtual Memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)* (May 1995).

[48] GANDHI, J., HILL, M. D., AND SWIFT, M. M. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43nd Annual International Symposium on Computer Architecture (ISCA-43)* (June 2016).

[49] GHOSE, S., BOROUMAND, A., KIM, J. S., GÓMEZ-LUNA, J., AND MUTLU, O. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development 63*, 6 (Aug. 2019), 3:1–3:19.

[50] GINGELL, R. A., MORAN, J. P., AND SHANNON, W. A. Virtual Memory Architecture in SunOS. In *Proceedings of the 1987 Summer USENIX Technical Conference (USENIX Summer '87)* (June 1987).

[51] GOOCH, R., AND ENBERG, P. Overview of the Linux Virtual File System. https://www.kernel.org/doc/html/next/filesystems/vfs.html, Feb. 2024.

[52] GOOGLE CLOUD. Memory-optimized machine family for Compute Engine. https://cloud.google.com/compute/docs/memory-optimized-machines, Apr. 2024.

[53] GOSAKAN, K., HAN, J., KUSZMAUL, W., MUBAREK, I. N., MUKHERJEE, N., SRIRAM, K., TAGLIAVINI, G., WEST, E., BENDER, M. A., BHATTACHARJEE, A., CONWAY, A., FARACH-COLTON, M., GANDHI, J., JOHNSON, R., KANNAN, S., AND PORTER, D. E. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)* (Mar. 2023).

[54] GRAY, C., CHAPMAN, M., CHUBB, P., MOSBERGER-TANG, D., AND HEISER, G. Itanium – A System Implementor's Tale. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX '05)* (Apr. 2005).

[55] GREGG, B. CPU Flame Graphs. https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html, Aug. 2021.

[56] GUPTA, S., BHATTACHARYYA, A., OH, Y., BHATTACHARJEE, A., FALSAFI, B., AND PAYER, M. Rebooting Virtual Memory with Midgard. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA-48)* (June 2021).

[57] HOANG, G., BAE, C., LANGE, J., ZHANG, L., DINDA, P., AND JOSEPH, R. A Case for Alternative Nested Paging Models for Virtualized Systems. *IEEE Computer Architecture Letters 9*, 1 (Jan. 2010), 17–20.

[58] HPC CHALLENGE BENCHMARK. RandomAccess: GUPS (Giga Updates Per Second). https://hpcchallenge.org/projectsfiles/hpcc/RandomAccess.html, Aug. 2022.

[59] HUANG, J., QURESHI, M. K., AND SCHWAN, K. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)* (June 2016).

[60] HUCK, J., AND HAYS, J. Architectural Support For Translation Table Management In Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA-20)* (May 1993).

[61] JALALIAN, S., PATEL, S., HAJIDEHI, M. R., SELTZER, M., AND FEDOROVA, A. ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC '24)* (July 2024).

[62] JIA, W., ZHANG, J., SHAN, J., DU, Y., DING, X., AND XU, T. HugeGPT: Storing Guest Page Tables on Host Huge Pages to Accelerate Address Translation. In *Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT'23)* (Oct. 2023).

[63] KANELLOPOULOS, K., BERA, R., STOJILJKOVIC, K., BOSTANCI, F. N., FIRTINA, C., AUSAVARUNGNIRUN, R., KUMAR, R., HAJINAZAR, N., SADROSADATI, M., VIJAYKUMAR, N., AND MUTLU, O. Utopia: Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-56)* (Oct. 2023).

[64] KANELLOPOULOS, K., SGOURAS, K., BOSTANCI, F. N., KAKOLYRIS, A. K., KONAR, B. K., BERA, R., SADROSADATI, M., KUMAR, R., VIJAYKUMAR, N., AND MUTLU, O. Virtuoso: Enabling Fast and Accurate Virtual Memory Research via an Imitation-based Operating System Simulation Methodology. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)* (Mar 2025).

[65] KARAKOSTAS, V., GANDHI, J., AYAR, F., CRISTAL, A., HILL, M. D., MCKINLEY, K. S., NEMIROVSKY, M., SWIFT, M. M., AND ÜNSAL, O. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42rd Annual International Symposium on Computer Architecture (ISCA-42)* (June 2015).

[66] KOPYTOV, A. SysBench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench/tree/1.0.20, Apr. 2020.

[67] KRUEGER, K., LOFTESNESS, D., VAHDAT, A., AND ANDERSON, T. Tools for the Development of Application Specific Virtual Memory Management. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)* (Sept. 1993).

[68] KWON, O., LEE, Y., PARK, J., JANG, S., TAK, B., AND HONG, S. Distributed Page Table: Harnessing Physical Memory as an Unbounded Hashed Page Table. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-57)* (Oct. 2024).

[69] KWON, Y., YU, H., PETER, S., ROSSBACH, C. J., AND WITCHEL, E. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (Nov. 2016).

[70] LEVY, H. M., AND LIPMAN, P. H. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer 15*, 3 (Mar. 1982), 35–41.

[71] LI, H., BERGER, D. S., HSU, L., ERNST, D., ZARDOSHTI, P., NOVAKOVIC, S., SHAH, M., RAJADNYA, S., LEE, S., AGARWAL, I., HILL, M. D., FONTOURA, M., AND BIANCHINI, R. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)* (Mar. 2023).

[72] Memory Type Range Registers in Linux Kernel. https://docs.kernel.org/arch/x86/mtrr.html.

[73] Paravirtualization in Linux Kernel. https://docs.kernel.org/virt/paravirt_ops.html.

[74] Page Attribute Table in Linux Kernel. https://docs.kernel.org/next/x86/pat.html.

[75] pv_ops.mmu in linux kernel. https://elixir.bootlin.com/linux/v5.15/source/arch/x86/kernel/paravirt.c#L290.

[76] MAGNUSSON, P., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. Simics: A Full System Simulation Platform. *IEEE Computer 35*, 2 (Feb. 2002), 50–58.

[77] MARGARITOV, A., USTIUGOV, D., BUGNION, E., AND GROT, B. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)* (Oct. 2019).

[78] MARUF, H. A., WANG, H., DHANOTIA, A., WEINER, J., AGARWAL, N., BHATTACHARYA, P., PETERSEN, C., CHOWDHURY, M., KANAUJIA, S., AND CHAUHAN, P. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)* (Mar. 2023).

[79] MERRIFIELD, T., AND TAHERI, H. R. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)* (Apr. 2016).

[80] NAI, L., XIA, Y., TANASE, I. G., KIM, H., AND LIN, C.-Y. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of SC15: The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)* (Nov. 2015).

[81] NAVARRO, J., IYER, S., DRUSCHEL, P., AND COX, A. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)* (Dec. 2002).

[82] Paging in OSDev Wiki. https://wiki.osdev.org/Paging.

[83] PAGH, R., AND RODLER, F. F. Cuckoo Hashing. *Journal of Algorithms 51*, 2 (May 2004), 122–144.

[84] PANWAR, A., BANSAL, S., AND GOPINATH, K. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)* (Apr. 2019).

[85] PANWAR, A., PRASAD, A., AND GOPINATH, K. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)* (Mar. 2018).

[86] PARK, C. H., HEO, T., JEONG, J., AND HUH, J. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA-44)* (June 2017).

[87] PARK, C. H., VOUGIOUKAS, I., SANDBERG, A., AND BLACK-SCHAFFER, D. Every Walk's a Hit: Making Page Walks Single-Access Cache Hits. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)* (Feb. 2022).

[88] PHAM, B., BHATTACHARJEE, A., ECKERT, Y., AND LOH, G. H. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proceedings of the 20th IEEE International Symposium on High-Performance Computer Architecture (HPCA-20)* (Feb. 2014).

[89] PHAM, B., VAIDYANATHAN, V., JALEEL, A., AND BHATTACHARJEE, A. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)* (Dec. 2012).

[90] PIGGIN, N. page table iterators. https://lwn.net/Articles/124037/, Feb. 2005.

[91] RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOLOSKY, W., AND CHEW, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)* (Oct. 1987).

[92] RAYBUCK, A., STAMLER, T., ZHANG, W., EREZ, M., AND PETER, S. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)* (Oct. 2021).

[93] REN, X. J., RODRIGUES, K., CHEN, L., VEGA, C., STUMM, M., AND YUAN, D. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)* (Oct. 2019).

[94] RODRIGUES, A., HEMMERT, K. S., BARRETT, B. W., KERSEY, C., OLDFIELD, R., WESTON, M., RIESEN, R., COOK, J., ROSENFELD, P., COOPER-BALIS, E., AND JACOB, B. The Structural Simulation Toolkit. *ACM SIGMETRICS Performance Evaluation Review 38*, 4 (Mar. 2011), 37–42.

[95] RYOO, J. H., GULUR, N., SONG, S., AND JOHN, L. K. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA-44)* (June 2017).

[96] SHARMA, D. D. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy. *IEEE Micro 43*, 2 (Mar. 2023), 99–109.

[97] SHUTEMOV, K. A. mm: convert generic code to 5-level paging. https://github.com/torvalds/linux/commit/c2febafc67734a62196c1b9dfba926412d4077ba, Mar. 2017.

[98] SKARLATOS, D., KOKOLIS, A., XU, T., AND TORRELLAS, J. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)* (Mar. 2020).

[99] STOJKOVIC, J., MANTRI, N., SKARLATOS, D., XU, T., AND TORRELLAS, J. Memory-Efficient Hashed Page Tables. In *Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA-29)* (Feb. 2023).

[100] STOJKOVIC, J., SKARLATOS, D., KOKOLIS, A., XU, T., AND TORRELLAS, J. Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)* (Feb. 2022).

[101] SUCHY, B., CAMPANONI, S., HARDAVELLAS, N., AND DINDA, P. CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)* (June 2020).

[102] SUCHY, B., GHOSH, S., KERSNAR, D., CHAI, S., HUANG, Z., NELSON, A., CUEVAS, M., BERNAT, A., CHAUDHARY, G., HARDAVELLAS, N., CAMPANONI, S., AND DINDA, P. CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)* (Feb. 2022).

[103] TABATABAI, B., SORENSON, J., AND SWIFT, M. M. FBMM: Making Memory Management Extensible With Filesystems. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC '24)* (July 2024).

[104] TALLURI, M., HILL, M. D., AND KHALID, Y. A. A New Page Table for 64-bit Address Spaces. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Dec. 1995).

[105] TORVALDS, L. Re: [Lse-tech] Re: 10.31 second kernel compile. https://yarchive.net/comp/linux/page_tables.html, Mar. 2002.

[106] VOUGIOUKAS, I. How about a short walk? https://community.arm.com/arm-research/b/articles/posts/how-about-a-short-walk, Mar. 2022. ARM Blogs.

[107] WOODRUFF, J., WATSON, R. N. M., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA-41)* (June 2014).

[108] YAN, Z., LUSTIG, D., NELLANS, D., AND BHATTACHARJEE, A. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)* (Apr. 2019).

[109] YANIV, I., AND TSAFRIR, D. Hash, Don't Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '16)* (June 2016).

[110] YE, H. Introduction to 5-Level Paging in 3rd Gen Intel Xeon Scalable Processors with Linux. https://lenovopress.lenovo.com/lp1468.pdf, May 2021.

[111] ZHANG, J., JIA, W., CHAI, S., LIU, P., KIM, J., AND XU, T. Direct Memory Translation for Virtualized Clouds. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)* (Apr. 2024).

[112] ZHOU, D., FAN, B., LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)* (Dec. 2013).

[113] ZUO, P., HUA, Y., AND WU, J. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)* (Oct. 2018).

Tiered Memory Management Beyond Hotness

Jinshu Liu Hamid Hadian Hanchen Xu Huaicheng Li
Virginia Tech

Abstract

Tiered memory systems often rely on access frequency ("hotness") to guide data placement. However, hot data is not always performance-critical, limiting the effectiveness of hotness-based policies. We introduce amortized offcore latency (AOL), a novel metric that precisely captures the true performance impact of memory accesses by accounting for memory access latency and memory-level parallelism (MLP). Leveraging AOL, we present two powerful tiering mechanisms: Soar, a profile-guided allocation policy that places objects based on their performance contribution, and Alto, a lightweight page migration regulation policy to eliminate unnecessary migrations. Soar and Alto outperform four state-of-the-art tiering designs across a diverse set of workloads by up to 12.4×, while underperforming in a few cases by no more than 3%.

1 Introduction

Driven by the growing demands of memory-intensive workloads, such as graph processing and machine learning, tiered memory architectures that integrate a **fast-tier** (*e.g.*, DRAM) and **slow-tier** (*e.g.*, CXL memory) are becoming standard in cloud datacenters [1–5]. While this approach improves memory capacity scaling, it also introduces significant performance challenges. Effective data tiering is critical to mitigating the 2–3× performance disparity between tiers [6–12].

Existing tiering designs are grounded in the assumption that frequently-accessed ("hot") data is more performance-critical than cold data and should reside in the fast-tier. Thus, tiered memory management primarily focuses on hotness tracking, memory allocation, and migration policies to detect, allocate, and relocate hot data across tiers efficiently [4, 13–29].

We argue that hot data is not always performance-critical and can reside in the slow-tier without degrading performance (§2.1). In modern out-of-order CPU designs, latency mitigation techniques, such as memory-level parallelism (**MLP**), obscure the true cost of memory accesses [13, 30–32]. Not all memory accesses contribute equally to performance (vary by 4×, §3); overlapping requests (high MLP) often mask slow-tier latency penalties, leading to less pronounced slowdowns.

Although MLP is a well-established concept within the architecture community [30–32], its implications for tiered memory management have been largely overlooked. Prior classification efforts across objects, pages, and data structures [13, 19, 33, 34] often implicitly reflect the effects of MLP through coarse heuristics or indirect indicators of memory access costs. However, they do not *explicitly* model or quantify MLP impact. What remains missing is a principled, accurate, and MLP-aware performance metric that enables more effective, performance-driven tiering policies across online and offline scenarios, and generalizes to diverse workloads.

Existing tiering systems also suffer from heavyweight and imprecise hotness sampling and page migration mechanisms. Two key limitations are prevalent [1, 4, 16, 17, 19, 21–24, 35]: **(a)** *Suboptimal data placement.* Existing coarse-grained allocation policies prioritize fast-tier placement for newly allocated data, but under fast-tier pressure, performance-critical data is often displaced to the slow-tier, necessitating costly migrations later to correct the placement errors; **(b)** *Excessive migration overhead.* Existing systems often employ aggressive migration policies, incurring substantial overhead by frequently relocating non-critical pages. This overhead can erode or negate the performance benefits of tiering (§2.1).

We propose **Amortized Offcore Latency (AOL)**, a novel performance metric that accurately quantifies the performance impact of memory accesses by integrating memory latency and MLP. While latency measures the impact of individual memory requests, it does not capture the latency-masking effects of MLP. By considering both factors, AOL, expressed as "`Latency/MLP`" combined with CPU stalls, offers a more precise representation of the true performance contribution of memory accesses (validated across 56 workloads, §3).

We leverage AOL to redesign memory allocation and migration policies, introducing two novel tiering mechanisms: a static memory allocation policy, **Soar**, and a dynamic page migration regulation policy, **Alto**. Soar employs AOL-based profiling to rank objects by assessing their accumulative contributions to application performance. High-ranking objects are placed in the fast-tier, achieving near-optimal placement while eliminating runtime migration overhead. Alto adaptively regulates page migrations based on AOL, ensuring that only performance-critical pages are promoted, regardless of their hotness. Alto seamlessly integrates with four representative tiering systems with minimal code changes, including TPP [4], Nomad [22], Linux NUMA Balancing Tiering (NBT) [36–38], and Colloid [23].

We evaluate Soar and Alto across a range of realistic

graph, cloud, and HPC workloads on both NUMA and real CXL platforms, varying fast-to-slow tier ratios and bandwidth contention levels. SOAR outperforms Nomad, NBT, Colloid, and TPP by 14–547%, 4–79%, -1–68%, and 31–1242%, respectively; ALTO improves performance by -2–81%, 1–31%, -3–18%, and 2–471%. Negative improvements indicate that SOAR/ALTO underperform relative to baselines in a few cases (5 out of 182 in total). While SOAR and ALTO achieve strong results broadly, their performance gains are less pronounced under high bandwidth contention due to AOL inflation from queuing delays. Raising AOL thresholds can restore their performance gains but requires contention-aware tuning. We highlight this to clarify the scope of our approach and leave AOL tuning as future work.

In summary, we make the following contributions:

- We quantitatively demonstrate that hotness is an unreliable proxy for performance-criticality: the performance impact of memory accesses can vary by up to 4× across workloads.
- We introduce AOL, a performance metric that combines memory access latency and MLP, and leverages CPU stall cycles to accurately estimate tiered memory performance.
- We propose AOL-powered memory management policies: SOAR for near-optimal data placement and ALTO for adaptive migration control.
- We open-source SOAR and ALTO at https://github.com/MoatLab/SoarAlto.

The rest of the paper is organized as follows: §2 covers background and motivation, §3 introduces the AOL metric and predictor, §4–6 present SOAR, ALTO, and evaluation, followed by related work (§7) and conclusion (§8).

2 Background and Motivation

Hotness implicitly assumes that all memory accesses contribute equally to application performance, ignoring variations introduced by loaded latency and CPU-side latency-mitigation techniques [39]. A key dimension behind this disconnect is **memory-level parallelism (MLP)**. MLP quantifies the number of outstanding memory requests awaiting completion by the memory controller. It plays a critical role in reducing CPU stall time [30]. Under high-MLP access patterns, such as array traversals, the CPU can issue many concurrent requests to minimize stalls. In contrast, serialized access patterns, as in pointer-chasing with depedent requests, exhibit low MLP and expose latency more directly, leading to greater slowdowns [13, 40]. In tiered memory systems, this implies that placing hot data in the fast-tier does not guarantee performance gains.

2.1 The Case Against Hotness-based Tiering

We design a microbenchmark to quantify the performance impact of different memory access patterns and illustrate how hotness-based tiering policies can lead to incorrect page migration decisions and performance degradation. Later, we show these findings extend to real-world applications with more complex memory access behaviors in §6.

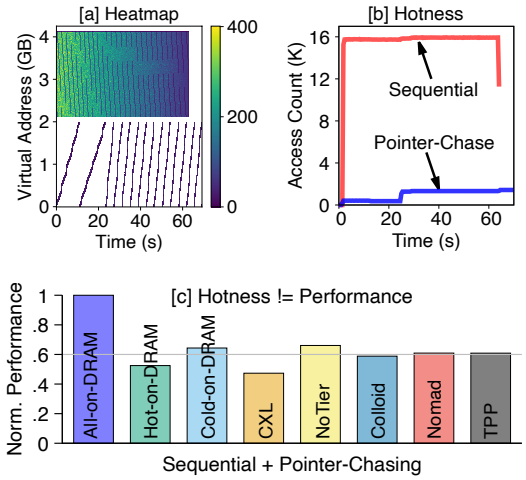


Figure 1: **Hotness vs. performance benchmark (§2.1).** *(a) presents the heatmap. (b) shows the access rates over time. (c) compares the performance of different tiering strategies (higher is better), showing that placing hot pages in the fast-tier can degrade performance when they are not performance-critical.*

The benchmark comprises two types of memory accesses: one thread performs sequential reads (high MLP, "hot", termed as **"seq"**), while the other executes pointer-chasing operations (low MLP, "cold", termed as **"pc"**). Each thread operates on a dedicated 2GB buffer. The "pc" thread issues 4 billion load instructions, while "seq" thread issues 26 billion, resulting in comparable runtimes for both (Figure 1a&b). This design prevents either thread from dominating overall workload performance in a tiered memory setup. The total runtime of both threads serves as the application's performance metric.

Figure 1a presents the heatmap highlighting the contrasting behaviors of the two memory access patterns. The sequential region (top half) demonstrates high memory access activity due to parallel memory reads (measured MLP=7 precisely), while the pointer-chasing region (bottom half) exhibits sparse memory accesses, reflecting its serialized nature (MLP=1). The difference is further quantified in Figure 1b, where "seq" pages are 13.6× hotter than "pc" pages on average.

Hotness-based tiering policies prioritize placing hot pages (in "seq") in the fast-tier, while cold pages (in "pc") are relegated to the slow-tier. Figure 1c shows the limitations of this approach. All the performance results are normalized to the fast-tier-only configuration (`All-on-DRAM`), where higher values represent better performance.

❶ Placing "seq" pages in the fast-tier and "pc" pages in the slow-tier (`Hot-on-DRAM`) degrades the performance to 52.4% of `All-on-DRAM`, nearly doubling the runtime.

❷ Placing cold "pc" pages in the fast-tier and hot "seq" pages in the slow-tier (`Cold-on-DRAM`) achieves a 34% performance gain over the "ideal" hotness-based placement in ❶.

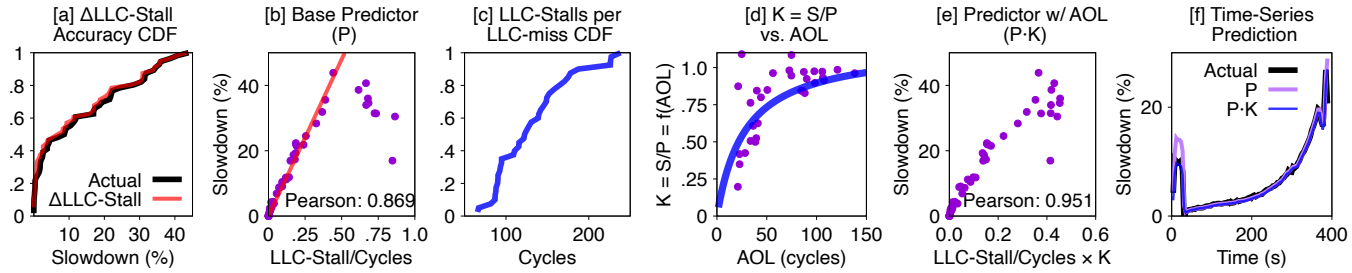


Figure 2: **AOL-based performance prediction (§3).** *(a) shows that LLC-Stalls effectively captures workload performance slowdowns on the slow-tier. (b) presents the base predictor (P) based on LLC-Stalls, but tends to overestimate slowdowns for high-MLP workloads. (c) reveals the average stalls per LLC-miss can vary by 4× across workloads. (d) models the workload-specific correction factor K as a function of AOL; the blue curve fits the observed hyperbolic relationship. (e) integrates K into the base predictor, yielding an AOL-based predictor that significantly improves prediction accuracy. (f) demonstrates that the AOL-based predictor generalizes to fine-grained time-series prediction.*

❸ When all pages are placed in the slow-tier (`CXL` bar), the performance is 47.4%, only slightly worse than ❶. Despite lower access frequency, the serialized nature of "pc" accesses dominates workload performance.

❹ State-of-the-art tiering designs such as Colloid, Nomad, and TPP underperform the `Cold-on-DRAM` setup, despite employing various optimizations.

❺ State-of-the-art also underperforms NoTier, a baseline that relies solely on the first-touch allocation policy without proactive page migrations. They fall behind NoTier by 12–14% and trail the ideal `All-on-DRAM` by 40%. This demonstrates that tiering can degrade performance due to the migration overhead and failure to identify truly performance-critical pages, which is not uncommon in real-world (§6).

The results highlight the inherent limitations of frequency-based hotness metrics and the corresponding hotness-driven tiering policies, where both incorrect data placement and migration overhead lead to suboptimal system performance.

3 Memory Performance Prediction

In this section, we define the Amortized Offcore Latency (AOL) metric for estimating the performance impact of slow-tier accesses, and demonstrate that AOL is accurate and adaptive across workloads at fine granularity.

3.1 Relating Slow-tier Performance to CPU Stalls

We begin with an offline slowdown analysis of 56 workloads from SPEC CPU 2017 [41] and GAPBS [42]. We measure the slowdown of each workload on the slow-tier compared to the fast-tier, and collect key CPU performance counters (Table 1) for intra- and inter-workload analysis.

We find that *performance degradation on the slow-tier is predominantly caused by increased CPU stalls due to LLC misses*, which we refer to as `LLC-Stalls` (or s_{LLC} for simplicity) [11, 43]. The slowdown (S) can be approximated as: $S = \frac{\Delta c}{c} \approx \frac{\Delta s_{LLC}}{c}$, where c is the number of CPU cycles on fast-tier, and Δs_{LLC} is `LLC-Stalls` increase on the slow-tier.

Table 1: **Intel PMU counters for AOL predictor (§3).** `ORO` is short for `OFFCORE_REQUESTS_OUTSTANDING`. Requests are demand reads.

s_{LLC}	`CYCLE_ACTIVITY.STALLS_L3_MISS`, # of LLC stall cycles
c	`CPU_CLK_UNHALTED.THREAD`, # of cycles
A_1	`ORO.CYCLES_WITH_DEMAND_DATA_RD`, cycles w/ pending requests
A_2	`ORO.DEMAND_DATA_RD`, # of pending requests per cycle
A_3	`OFFCORE_REQUESTS.DEMAND_DATA_RD`, # of requests to uncore

It is important to distinguish `LLC-Stalls` from LLC misses. While LLC misses count how often memory accesses reach fast-tier/slow-tier, `LLC-Stalls` measure the *actual stalled CPU cycles* waiting on such memory accesses. Thus, `LLC-Stalls` offer a more direct and actionable signal of slow-tier impact on application performance. Due to the 2–3× latency increase of the slow-tier [8, 22], each LLC miss results in more stall cycles, making `LLC-Stalls` a natural proxy for slowdown. Figure 2a shows the CDFs of actual and predicted slowdowns using $\frac{\Delta s_{LLC}}{c}$ for all the 56 workloads. Estimated slowdowns deviate by less than 4% from measured values, confirming that the added CPU stalls induced by slow-tier accesses largely explain workload slowdowns.

3.2 LLC-Stalls for Performance Prediction

While $\frac{\Delta s_{LLC}}{c}$ accurately estimates slowdown, computing Δs_{LLC} requires measuring workload performance on both the fast-tier and the slow-tier, limiting its use to offline settings. To enable online prediction, we simplify it using the fast-tier metric $\frac{s_{LLC}}{c}$ based on the following observation.

Our analysis shows that Δs_{LLC} and s_{LLC} are strongly correlated: workloads with high `LLC-Stalls` on the fast-tier tend to incur proportionally more stalls on the slow-tier. This allows us to approximate $\Delta s_{LLC} \approx k \times s_{LLC}$, where k is a constant. Substituting into the slowdown formula, we get $S = \frac{\Delta s_{LLC}}{c} \approx k \times \frac{s_{LLC}}{c}$. We define the base predictor P as $P = \frac{s_{LLC}}{c}$. Figure 2b shows that S and P are strongly correlated across 85% of workloads, with a Pearson coefficient of 0.869. The red line shows the fitted model; purple dots are measured slowdowns offline.

3.3 AOL for Accurate Prediction

Amortized Offcore Latency (AOL). P is accurate for low-MLP workloads but fails to model high-MLP workloads (outliers in Figure 2b). Upon further investigation, P tends to overestimate slowdown for high-MLP workloads (>4 on our platform). This overestimation stems from its implicit assumption that all LLC misses equally contribute to CPU stalls, ignoring the latency-masking effect of MLP. Figure 2c shows that the average `LLC-Stalls` per LLC miss vary significantly across workloads, ranging from 60 to 240 cycles (a 4× difference). High-MLP workloads exhibit fewer stalls per miss, reflecting reduced sensitivity to slow-tier latency.

While MLP's conceptual impact on performance is intuitive, its *quantitative* effect is much harder to model. Naively integrating MLP into the predictor (*e.g.*, $\frac{P}{\texttt{MLP}}$) yields poor correlation with slowdown. Moreover, MLP alone is insufficient for modeling slowdown. Its latency-masking benefits diminish as memory latency increases. To address this, we define $\texttt{AOL} = \frac{\texttt{Latency}}{\texttt{MLP}}$ and use it to enhance the predictor.

AOL-Based Prediction Model. We use AOL to refine the base predictor P by regulating its overestimation for high-MLP workloads. Specifically, we model slowdown as $S = P \times K$, where K **is a function of AOL that quantifies how MLP amortizes the base predictor's overestimation.** K-vs.-AOL is derived via offline cross-workload modeling following an empirical approach. For each workload, given the measured slowdown S and $P = \frac{s_{LLC}}{c}$, we compute $K = \frac{S}{P}$, resulting in the purple data points in Figure 2d (X-axis is AOL and Y-axis is K). The nonlinear relationship between K and AOL indicates that MLP does not scale down P by a constant factor. This aligns with intuition, doubling MLP from 2 to 4 does not yield a 2× performance gain, as modern CPUs employ various latency-hiding optimizations that complicate MLP's direct impact on slowdown.

Observing that K follows a hyperbolic trend with asymptotic growth behavior [44], we fit the curve using $K = f(\texttt{AOL}) = \frac{1}{a+\frac{b}{\texttt{AOL}}}$, where a and b are constants. The resulting fit is shown as the blue curve in Figure 2d. Importantly, a and b are hardware-dependent (*e.g.*, CPU and memory) but workload-independent. They can be calibrated offline using microbenchmarks with extreme access patterns (*e.g.*, sequential vs. pointer-chasing), which represent two ends of the MLP spectrum, as discussed in §2.1. Users do not need to repeat the extensive benchmarking process we performed to profile and model $K = f(\texttt{AOL})$. This makes the model easy to deploy across platforms, enabling fast and accurate online prediction with minimal profiling overhead. Figure 2e shows that AOL significantly improves prediction fidelity, achieving a Pearson correlation of 0.951 (closer to 1 indicates stronger linear relationship). While our model is not perfect, evidenced by an outlier in the bottom right of Figure 2e, it generalizes well across diverse workloads (§6). We leave more accurate modeling to future work [43].

Lightweight Measurement. All components needed for AOL and the AOL-based predictor are derived from just four hardware counters (Table 1): A_1, A_3, c, and s_{LLC}.

❶ $\texttt{Latency} = \frac{A_2}{A_3}$, where A_2 is the accumulative number of inflight requests per cycle and A_3 is total requests to uncore.
❷ $\texttt{MLP} = \frac{A_2}{A_1}$, where A_1 counts cycles with ≥1 inflight request.
❸ $\texttt{AOL} = \frac{\texttt{Latency}}{\texttt{MLP}} = \frac{A_1}{A_3}$.
❹ Base predictor $P = \frac{s_{LLC}}{c}$ measures stall pressure.
❺ We compute $K = \frac{S}{P}$ and model it with $K = f(\texttt{AOL}) = \frac{1}{a+\frac{b}{\texttt{AOL}}}$.
❻ Final predictor is

$$S = P \times K \approx \frac{s_{LLC}}{c} \times \frac{1}{a+\frac{b}{\texttt{AOL}}}$$

Here, AOL captures how MLP and memory access latency jointly shape performance.

AOL vs. Slowdown. We now present the properties of $K = f(\texttt{AOL})$ and how it relates to slowdown via $S = P \times K$, by analyzing the (blue) curve in Figure 2d. The observed AOL range spans mostly (0, 130] cycles, with K values in mainly (0, 1] on our testbed (§6). This formulation captures how AOL reflects the impact of CPU stalls on slowdown: as AOL increases (*i.e.*, high latency or low MLP), K approaches its upper bound of 1, making S nearly linear with P. Conversely, small AOL values (*e.g.*, high MLP) indicate that most stalls are masked, yielding smaller slowdowns with K closer to 0.

The curve also reveals diminishing returns at high AOL. For 70% of workloads with AOL below 90 cycles, K increases steeply from 0.2 to 0.8, indicating that P requires significant correction only when AOL is low. A small K (*e.g.*, 0.5) implies that P must be scaled down by 2× to match observed slowdown. In contrast, the remaining 30% of workloads with higher AOL require less than 20% adjustment, suggesting that raw stall time already tracks slowdown closely. When memory bandwidth is unconstrained and latency is stable, MLP becomes the dominant factor in AOL and thus drives K, quantifying its direct influence on slowdown.

AOL remains effective even under bandwidth contention. Under bandwidth pressure, queuing delays inflate latency, which raises AOL. This behavior reflects the growing performance cost, for both latency- and bandwidth-bound workloads.

Time-Series Prediction. Beyond workload-level modeling, the AOL-based predictor supports fine-grained, time-series slowdown prediction. This enables accurate performance estimation over short execution intervals, essential for adaptive, online/offline tiered memory management (§4–§5). Figure 2f shows the prediction results for a graph workload (`tc-twitter`). Compared to the base predictor P (pink), which fails to capture the dynamics, especially in the first ~50s, the AOL-based predictor $P \cdot K$ (blue) closely matches the actual slowdown (black), demonstrating its effectiveness for interval-level prediction.

Next, we show how AOL and its predictors can be used to guide data placement and migration in tiered memory systems.

4 SOAR: Rank-based Static Object Allocation

Existing tiered systems rely on the first-touch policy, supplemented by LRU-based page reclamation to maximize fast-tier usage [4, 45]. We argue *tiered memory allocation should prioritize performance-critical objects for fast-tier placement.*

We seek a **near-optimal** initial object placement strategy, *eliminating* the need for costly page migrations. Achieving this requires capturing object-level performance contributions across diverse object types and temporal dynamics. To this end, we introduce **SOAR**[1], an AOL-driven profiling-guided memory allocation policy based on object rankings according to their accumulative contribution to workload performance.

While AOL-based prediction (§3) is effective at the workload level, it falls short for individual objects due to the semantic gap between architectural events (Table 1) and object-level memory accesses. We develop a novel object-level profiling algorithm that refines AOL-based performance prediction to operate at object granularity. The key insight is to distribute CPU stalls across objects proportionally to their relative access frequencies based on the observed MLP and latencies, thereby approximating each object's performance impact to application performance accurately.

4.1 Object-Level Performance Profiling

Figure 3 (❶–❽) illustrates SOAR's profiling workflow, which periodically collects and processes three types of metrics: object metadata via object tracking, memory accesses via PEBS-based LLC-miss sampling, and temporal performance via AOL-based prediction. SOAR's key innovation lies in associating these data streams to derive a quantified per-object performance impact (a "score") for ranking. SOAR profiler runs the workload *once* on the fast-tier to gather all required metrics with minimal performance overhead.

4.1.1 Object Profiling

We now describe the three data flows used for SOAR's object profiling in detail.

Object Tracking/Flow (F_O, ❶→❷): We track object metadata to analyze usage patterns. Using `LD_PRELOAD`, we intercept (de)allocations via `malloc()/free()` and `mmap()/munmap()`. For each object, we record its lifespan ($[T_{\text{alloc}}, T_{\text{free}}]$), virtual address (`vaddr`), size, allocation `type` such as `malloc()` or `mmap()`. We group objects by call chain via `backtrace()`, treating those with identical call stacks as the same object type, as they originate from the same code path and share access patterns. (De)allocations are then matched by `vaddr`, with each object represented as a five-element tuple, forming the object flow (F_O).

Memory Access Tracking/Flow (F_M, ❸→❹): We use Intel Processor Event-Based Sampling (PEBS) to track the temporal and spatial distribution of memory accesses. Specifically, we sample LLC misses, recording the access timestamp (T) and virtual address (`vaddr`). High-fidelity sampling is unnecessary: a low sampling rate (*e.g.*, 3000) suffices, imposing negligible overhead and avoiding timing skew, which is critical for profiling short-lived objects. Each PEBS sample forms an entry in the memory access flow (F_M).

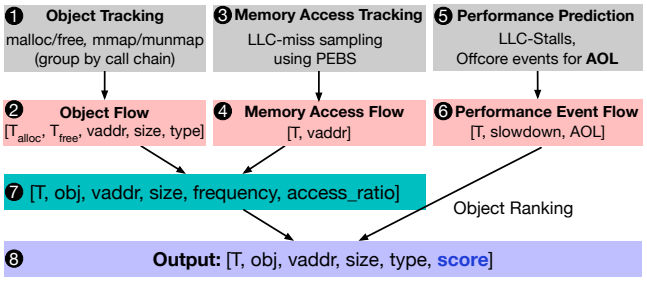


Figure 3: **SOAR profiling (§4.1).** *SOAR tracks multiple flows of information to derive object rankings based on performance impact.*

Performance Analysis/Flow (F_P, ❺→❻): We leverage AOL-based prediction to estimate memory access performance impact (§3). Performance is sampled periodically over workload execution (*e.g.*, every one second, configurable). Each sample forms an entry in the performance flow (F_P), including the timestamp, predicted performance, and AOL.

4.1.2 Unifying Object Flows

Next, we unify the three flows (F_O, F_M, F_P) for analyzing object characteristics and convert them into a comprehensive per-object performance profile.

We first merge F_O and F_M to associate memory accesses with objects. If the timestamp and address of a memory access in F_M fall within the lifecycle and address range of an object in F_O, the memory access is attributed to the corresponding object. All memory accesses in F_M are examined and matched to objects in F_O, constructing a memory access time-series flow (T_M) for each object (O).

For each object, the time-series flow (T_M, ❼) is generated over its lifetime. Each entry in T_M includes the timestamp, ID, address, size, and *access frequency* between the current and previous timestamps. After constructing all T_M data flows, the number of memory accesses (c) to each object during each profiling period can be computed. The memory access ratio (R) for each object is defined as $R = \frac{c}{\sum_i c_i}$. It represents the weight of an object's memory accesses relative to the total memory accesses, which will be used to assign object-level performance slowdowns (§4.2).

We then merge T_M with F_P to associate the predicted performance metrics with each object. The time-series predicted performance is derived from the performance events flow (F_P), where the predicted performance and AOL are computed for each time interval, along with the timestamp, forming the time-series performance flow (T_P). By combining T_M and T_P (Algorithm 1, §4.2), the predicted performance and AOL are associated with live objects during each time

[1] SOAR stands for **S**tatic **O**bject **A**llocation based on **R**anking.

Algorithm 1 Object scoring (every profiling period)

Input: 1) Access ratio (R); 2) Predicted perf (p); 3) AOL (l)
Output: Object score (s).

1: `factor` = $F(l)$ ▷ *Decide perf scale-factor based on AOL*
2: **if** $R < R_{min}$ and $l < L_0$ **then**
3: $s = R \times p \times$ `factor` ▷ *Low-MLP object*
4: **else if** $R > R_{max}$ and $l < L_0$ **then**
5: $s = R \times p$ / `factor` ▷ *High-MLP object*
6: **else**
7: $s = R \times p$ ▷ *MLP=1, even hotness*
8: **end if**

period. This process results in a comprehensive time-series object flow/profile (T_O, ❽) constructed for each object.

4.2 Object Ranking

The object ranking process quantifies each object's cumulative contribution to workload performance over its lifetime. For all active objects during a profiling interval, and given the predicted workload performance (§3.3), the core challenge is attributing performance impact to individual objects. This is non-trivial because modern hardware does not provide mechanisms to directly measure per-object performance contributions. While the AOL-based predictor accurately estimates slowdown at the workload level, it does not bridge this granularity gap. Furthermore, current CPUs do not expose per-access CPU stall information, making fine-grained attribution infeasible. Soar employs a simple yet effective heuristic: it estimates relative object contributions based on MLP and access frequency, detailed in Algorithm 1.

In an extreme scenario with no memory overlapping effect during the time period (*i.e.*, MLP=1), the predicted performance slowdown can be distributed proportionally to the memory accesses (c) for each object. Let the predicted performance slowdown for the period be p. The score for each object is then computed as $p \times R$ (Lines 6–7). This is true because each memory access contributes equally to the overall predicted performance slowdown under MLP=1.

When memory overlapping effects are significant (high MLP), objects with a higher number of memory accesses dominate the memory overlapping behavior. Their performance contributions (and scores) should be amortized to account for their likely higher MLP compared to other objects. This scenario corresponds to Lines 4–5 in Algorithm 1. Similarly, for objects with lower MLP, their average per-access performance contribution is higher than that of objects with higher MLP. Therefore, their scores should be scaled up to reflect this increased performance contribution (Lines 2–3).

R_{min} and R_{max} are used to differentiate high- and low-frequency accessed objects. The "hot" objects are more likely to be affected by memory overlapping when AOL is low, while the scores of "cold" objects need to be scaled up. For example, Figure 1b illustrates an extreme case: the score of a pointer-chasing object should be higher than that of a sequential object due to the low AOL. L_0 is the threshold used to determine whether the current time period exhibits significant MLP. This threshold is derived from our microbenchmark results, which establishes the relationship between K and AOL in Figure 2d. For instance, when AOL is more than 100 cycles, K stabilizes to a constant, thus we set L_0 to be 100.

The scale factor $F(l)$ (Line 1) adjusts per-object performance estimates based on access frequency (§3.3), analogous to how K in Figure 2d amortizes latency at the workload level. We use AOL to determine this scaling factor. When AOL is low, we set the factor to 8, corresponding to the MLP of access patterns with high parallelism (*e.g.*, the sequential microbenchmark in §2, which has MLP=7). Dividing an object's access count by 8 effectively cancels out the masking effects of MLP. For objects with very low access frequency (*e.g.*, pointer-chasing), we amplify their estimated performance impact by multiplying their access count by a factor between 2 and 8, determined by a stepwise function of the observed AOL (higher AOL corresponds to a larger factor). We later show that a similar approach is also effective for regulating page migrations in §5.

Aggregating per-interval scores across all time intervals for each object is straightforward: Once a score (s) is assigned to an object at a given time interval, it is accumulated over the entire lifecycle of that object type. This approach aligns with the memory allocator's primary objective, minimizing performance degradation when allocating objects by ensuring that all performance contributions are accounted for. This contrasts with online tiering policies, which often prioritize recent accesses by giving them more weight [17, 19].

After the entire process is complete, each object type is assigned a score s. To account for varying object sizes, a unit score is introduced as $s' = \frac{s}{\texttt{sizeof}(O)}$. When comparing two objects with the same score s, the larger object is less valuable to place on fast-tier due to its lower unit score.

4.3 Object Allocation

Soar allocation decision is based on the rank of objects by their unit scores. It aims to place the top-N objects in fast-tier, where it tries to maximize N while ensuring that the total size of the top-N objects does not exceed fast-tier size. Since fast-tier size does not always match the total size of the top-N objects, we attempt to bind as many top-ranking objects to fast-tier as possible from a sorted list of objects $(O_1, \ldots, O_n)$ with unit scores $(s'_1, \ldots, s'_n)$ in descending order, where $s'_i \geq s'_{i+1}$. For short-lived objects that may be interleavingly allocated with others, the sum of their occupied size on the fast-tier is taken as the maximum of their individual sizes. If free space is insufficient to fully accommodate the next top-ranking object when the request arrives, Soar falls back to the first-touch approach: the object is placed in the fast-tier first and spills over to the slow-tier when the fast-tier becomes full.

The sorting order of objects by unit scores does not necessarily correspond to the order of their allocation requests. For instance, the $(k+1)^{\text{th}}$ ranked object may be allocated before some of the top-k objects. In such cases, pages of the $(k+1)^{\text{th}}$ object are demoted to the slow-tier until enough space becomes available in the fast-tier for the top-k objects. Page demotion is triggered only when space is insufficient for objects that's destined to stay in fast-tier, making the total demotion overhead low as it occurs rarely. Objects that are neither fully nor partially allocated to the fast-tier are allocated to the slow-tier.

We use `numa_alloc()` from `libnuma` to overload memory allocation functions and bind allocations to the fast-tier/slow-tier. Objects that can be flexibly placed on either tier retain their original allocation path without being overloaded. This approach requires no changes to application code, making SOAR non-intrusive and easy to use. To identify which allocation should be redirected, SOAR inspects the call chain at each allocation site to distinguish object types. It supports various languages (*e.g.*, C/C++, Python) and does not depend on specific memory allocators. For example, SOAR can also integrate with heterogeneous memory-aware allocators such as `memkind` [46, 47], which mitigates potential fragmentation for small objects, as `numa_alloc()` operates at page granularity.

4.4 Use Cases and Limitations

Modern applications such as graph processing, ML/AI, and HPC often pre-allocate objects that persist for extended periods, making them ideal candidates for SOAR. Although SOAR adopts static allocation based on offline profiling, it can be extended to support online profiling for long-running workloads. One approach is to use past profiling data to predict future object performance.

Although SOAR requires a single run of the workload on the fast-tier for profiling object scores, profile-guided optimization is a widely adopted practice for improving datacenter efficiency [48–50]. Another limitation is that the current SOAR ranking algorithm assumes uniform memory access distribution across each object, leaving room for future optimizations for objects with heterogeneous access patterns.

5 ALTO: AOL-based Adaptive Page Migrations

In this section, we show that AOL can also address a key bottleneck in existing tiering designs: excessive page migrations that disregard performance impact, leading to unnecessary overhead and degraded performance. By prioritizing the migration of performance-critical pages and filtering out less impactful ones, AOL improves overall tiering efficiency.

5.1 ALTO Overview

Existing tiering designs adopt aggressive page migration strategies: when a "hot" page is detected, it is immediately promoted to the fast-tier, either because space is available or by demoting cold pages to make room. This policy has several drawbacks. *(a)* Migrating hot but non-performance-critical pages yields no benefit, as these accesses do not induce CPU stalls. *(b)* Page migrations are long-latency, blocking operations that impose substantial overhead. Per our measurements, migrating a page takes on average 12µs, during which application threads are stalled if they access the migrating page. *(c)* This challenge is exacerbated on CXL, where the latency and bandwidth gap with DRAM is narrowing, making tiering overhead more pronounced. Consequently, many state-of-the-art tiering systems underperform even naive first-touch-based baselines due to excessive migration overhead. *(d)* Worse, performance-critical cold pages are often ignored by access-frequency heuristics, missing opportunities for performance gains. These limitations call for a fundamental reassessment of assumptions in current tiering policies.

Ideally, pages should be migrated only when they are truly performance-critical, and unnecessary migrations should be avoided when the workload is insensitive to slow-tier accesses.

To address these issues, we propose **ALTO**[2], an adaptive tiering orchestration policy that dynamically regulates page migration intensity. ALTO leverages the AOL metric to detect periods of high memory access overlap, during which slow-tier accesses have minimal performance impact. By filtering out non-critical migrations, ALTO reduces overhead and improves overall performance. ALTO is lightweight and easily integrates into existing tiering systems, enhancing efficiency without requiring major architectural changes.

Non-Goal: While integrating AOL to track per-page performance and design AOL-centric migration policies is promising, it presents unique challenges, particularly in estimating page-granular performance using coarse-grained counters (out of scope for this work). We plan to explore the broader tiering design space enabled by AOL in future work.

5.2 ALTO Design

Leveraging AOL, ALTO regulates page migrations when the overlapping effect of memory accesses (high MLP) is evident. In other words, we can use AOL to identify non-performance-critical periods and adjust the intensity of tiering operations accordingly. The detailed ALTO page migration regulation scheme is shown in Algorithm 2.

Let us first consider the case where memory bandwidth is not a bottleneck, so offcore latency remains stable and low, making MLP the dominant factor in AOL.

(a) Low AOL: Low AOL indicates high MLP, meaning memory latency is largely masked and potential slowdown is minimal. In this case, there is less need for page promotions. ALTO limits the rate of hot page detection, page migrations, or both, to reduce unnecessary overhead.

(b) High AOL: High AOL suggests serialized, latency-sensitive memory accesses where correct page promotions

[2] ALTO stands for **A**OL-based **L**ayered **T**iering **O**rchestration.

Algorithm 2 AOL-regulated page migrations (*e.g.*, every 1s)

Input: $\texttt{AOL}_{\texttt{low}} \leftarrow 40$, $\texttt{AOL}_{\texttt{high}} \leftarrow 100$ ▷ *Profiled offline*
Output: Adjusted page migration ratio (`Scale`).

```
1: Profile the current AOL (l)
2: if l ≤ AOL_low then
3:     Scale ← 0                          ▷ Disable page promotions
4: else if l ≥ AOL_high then
5:     Scale ← 1                          ▷ Enable all page promotions
6: else
7:     Scale ← F(l)                       ▷ Partial page promotions
8: end if
```

can yield significant performance gains. Alto responds by enabling more aggressive page detection and migration to alleviate critical bottlenecks.

Based on the above observations, Alto employs two AOL thresholds to guide page migration regulations: a lower bound ($\texttt{AOL}_{\texttt{low}}$) and an upper bound ($\texttt{AOL}_{\texttt{high}}$). When AOL falls below the lower bound, Alto limits or disables tiering operations to reduce overhead (Lines 2–3). When AOL exceeds the upper bound, Alto enables full-speed tiering operations (*i.e.*, no change to the default tiering migration policy, Lines 4–5).

Since the impact of AOL on performance follows a hyperbolic curve (Figure 2d), where small changes near the lower bound can lead to large performance differences (§3.3), a more fine-grained approach could involve dynamically adjusting tiering intensity based on the observed AOL value. This enables more nuanced and adaptive tiering decisions to better capture workload dynamics. We use a stepwise function to adjust page migration intensity. Specifically, Alto gradually reduces the page promotion rate as AOL decreases while AOL falls within $[\texttt{AOL}_{\texttt{low}}, \texttt{AOL}_{\texttt{high}}]$ (Lines 6–7). The function $F(l)$ (Line 7) mirrors the procedure used to determine the performance scale factor $F(l)$ in Algorithm 1 in Soar (§4).

The AOL thresholds in Alto are derived from the blue curve in Figure 2d, modeled as $K = f(\texttt{AOL}) = \frac{1}{a+\frac{b}{\texttt{AOL}}}$. We use an empirical approach based on two microbenchmarks representing extreme MLP cases: a pointer-chasing workload for low MLP and a sequential workload for high MLP (§2). These benchmarks yield low/high AOL values of 40/100 cycles and 25/95 cycles on our two experimental platforms. We conduct detailed sensitivity studies on these values in §6.

5.3 Alto Integration with Existing Tiering Systems

Alto can be seamlessly integrated into existing tiering systems, such as TPP [4], NBT [36, 37], Nomad [22], and Colloid [23], to enhance their efficiency and reduce overhead. This integration is straightforward as Alto builds upon their existing policies. Below, we provide a brief overview of how Alto can be incorporated into these systems.

Alto+TPP. TPP is a state-of-the-art tiering design for CXL, which adopts page reclamations for pages demotions and NUMA hinting faults for page promotions with a set of aggressive heuristics to identify hot pages. We implement Alto+TPP by constraining the page promotion rate proportionally to AOL based on offline-profiled thresholds.

To gradually reduce page promotion rate as AOL decreases, in our implementation, Alto+TPP periodically ignores certain potential promotion candidate pages. For instance, if we aim to allow 20% of TPP-identified candidate pages to be promoted, we allow the first two pages of every 10 pages to go through.

To monitor AOL, we utilize Linux `perf` to collect the CPU counters periodically (Table 1), *e.g.*, every 1s. Subsequently, we calculate AOL based on these counters, enabling us to dynamically adjust the page promotion rate based on the observed AOL. Our user-level tool is lightweight and imposes no additional overheads. The kernel side only involves ~30 LOC changes to page migration policies in the Linux memory subsystem. We use a default AOL sampling period of 1s for Alto. While lower sampling period possibly enables more fine-grained Alto-based migration regulations and better performance, we find that 1s is sufficient to capture AOL and workload dynamics.

Alto+NBT, Alto+Nomad, and Alto+Colloid. Unlike TPP, NBT, Nomad, and Colloid adopt less aggressive page migration strategies. They rely on NUMA hinting faults for tracking page accesses, which functions similarly to standard NUMA balancing (*i.e.*, AutoNUMA [51]). Their page scanning mechanism sequentially examines all Virtual Memory Areas (VMAs) in each process. During VMA scanning, the system sets each page's flag to `PAGE_NONE`. Subsequently, when a page is accessed, a minor page fault is triggered. If the page resides on a node other than its preferred node, it is marked as a candidate for migration to the preferred node.

In both NBT and Nomad, only pages located in the slow-tier are scanned, with the fast-tier serving as the preferred node for all pages. To optimize this process, we implement Alto by limiting the number of pages set to `PAGE_NONE` during periods of significant memory access overlapping (low AOL), effectively regulating promotions. Colloid samples pages in both tiers following the same mechanism. For Alto+Colloid, we only regulate page migrations from slow-tier to fast-tier.

6 Evaluation

Our evaluation seeks to answer three key questions: (1) How do Soar (§6.2–§6.4) and Alto (§6.5–§6.7) compare to state-of-the-art tiering policies? (2) How sensitive are their performance gains to AOL threshold choices? (§6.8) (3) How do they perform under bandwidth contention? (§6.9–§6.10)

6.1 Experimental Setup

We evaluate Soar and Alto on two platforms. The first is a CloudLab dual-socket Intel Skylake server (**"SKX"**) with two 10-core CPUs and 96 GB DDR4 DRAM per socket [52]. We emulate CXL by lowering the uncore frequency and

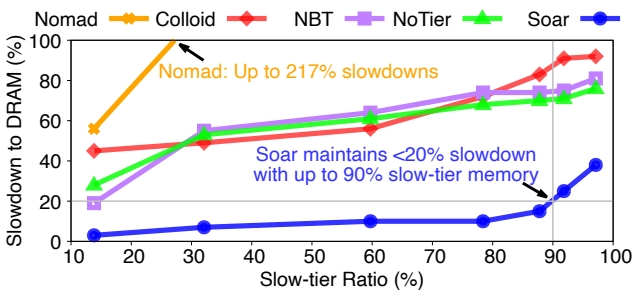


Figure 4: **Soar on SKX/NUMA (§6.2).** *Soar consistently outperforms all other schemes under bc-urand across various slow-tier ratios, whereas state-of-the-art approaches frequently underperform, even compared to NoTier, especially under higher slow-tier ratios.*

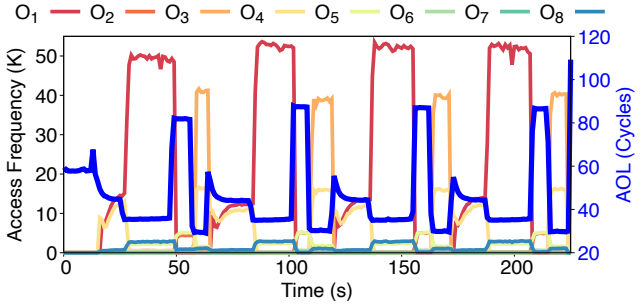


Figure 5: **Object-level accesses and AOL in Soar (§6.2).** *High object access frequency correspond to low AOL and cross-object access frequency correlates with MLP, which can be used to approximate object performance.*

disabling cores on one NUMA node, resulting in fast/slow-tier latencies of 90/190 ns (2.1×) and bandwidths of 49/17 GB/s. The second platform is a local Intel Sapphire Rapids server (**"SPR"**), with a 32-core CPU per socket, 192 GB DDR5 DRAM, and an ASIC-based 128 GB CXL memory expander (PCIe 5 ×8). Fast/slow-tier latencies are 114/271 ns (2.4×), and bandwidths are 218/26 GB/s.

We compare Soar and Alto against TPP [4], Nomad [22], NBT [36–38], Colloid [23], and a first-touch-only baseline ("NoTier"). NBT is the successor to AutoNUMA in Linux, with upstreamed optimizations from TPP. Colloid provides three implementations built on top of HeMem [17], Memtis [19], and TPP. We use the Colloid implementation built on top of TPP, which includes CXL-specific optimizations. **Performance is reported as slowdown relative to fast-tier-only (DRAM) performance**, which provides a fair and consistent comparison across various target systems. Lower slowdown (closer to 0) indicates better performance. None of the tiering systems outperform the fast-tier-only configuration.

Our workloads span graph analytics [42], machine learning [53], caching [54], and HPC [41], running under various fast/slow-tier ratios (mainly 10–90%, relative to workload's RSS). Each workload runs with 8 threads by default (unless otherwise noted), with bandwidth usage of 2.3–21 GB/s and RSS of 8–35 GB. We present detailed results for a few representative graph workloads and summarize the rest later.

6.2 Soar for Graph Processing

`bc-urand` is a betweenness centrality workload from the GAPBS benchmark suite [42], executed on a synthetic uniformly random undirected graph. We use the default configuration, which generates a graph with 134 million vertices and 2147 million edges. The algorithm estimates centrality by computing shortest paths from a subset of source vertices, resulting in irregular and sparse memory accesses. Its memory footprint is ~20 GB. The combination of large working set size (~17 GB) and random access patterns makes `bc-urand` a representative stress test for tiered memory systems. Figure 4 shows detailed Soar performance results compared to state-of-the-art tiering systems across various slow-tier ratios.

Table 2: **Soar object statistics and rankings (§6.2).** *The object information includes size, lifetime, access frequency, and computed Soar score. Rankings are provided based on three distinct criteria: First-touch (FT), Frequency (Freq), and Soar.*

#	Obj	Size	Time	Freq	Score	Ranking		
						FT	Freq	Soar
O_1	5fb2	536MB	139s	4.7M	$3.5e^{-8}$	8	1	1
O_2	6d68	536MB	208s	201K	$1.9e^{-8}$	6	6	2
O_3	6fe7	536MB	60s	1.3M	$1.8e^{-8}$	10	2	3
O_4	6d27	1073MB	208s	1.2M	$1.7e^{-8}$	5	3	4
O_5	b69c	1073MB	208s	420K	$1.4e^{-8}$	3	4	5
O_6	6cc3	536MB	208s	20.3K	$1.7e^{-9}$	4	7	6
O_7	6db6	536MB	208s	309K	$1.2e^{-9}$	7	8	7
O_8	b62e	17GB	223s	313K	$5.3e^{-10}$	2	5	8
O_9	5c24	327KB	14s	0	0	9	9	9
O_{10}	b5fb	1073MB	139s	0	0	1	10	10

Takeaway #1: *Soar outperforms all baselines under bc-urand across all slow-tier ratios. Soar maintains less than 20% slowdown even under 90% slow-tier memory, demonstrating robust performance under aggressive tiering conditions.*

In contrast, Nomad suffers up to 217% slowdown, and both NBT and Colloid degrade steadily (>60% slowdown) as the slow-tier ratio increases. Under high slow-tier ratios (>80%), all tiering baselines underperform NoTier by 10–20% due to excessive page migrations. Soar is the only system that outperforms NoTier consistently. We defer a detailed analysis of the inefficiencies in existing tiering designs to §6.5. These results highlight the effectiveness of Soar's performance-criticality-aware object allocation in maintaining good performance even under severe memory pressure.

Understanding Soar object rankings. Table 2 and Figure 5 summarizes the object-level statistics and rankings. The left five columns report each object's ID, address, size, lifetime, and access frequency. The "Score" column reflects the per-object unit-score computed using Algorithm 1, which determines Soar object rankings (Column 9). For comparison, we also include rankings based on first-touch (**"FT"**, Column 7) and frequency-only (**"Freq"**, Column 8) policies.

Table 3: **SOAR object placement (50% slow-tier ratio).**

	Fast-tier	Slow-tier
SOAR	O_1–O_8	O_8–O_{10}
FT	O_{10}, O_8	O_1–O_7, O_9
Freq	O_1, O_3–O_5, O_8	O_2, O_6–O_7, O_9–O_{10}

Object information is as follows: O_8 represents the graph constructed after reading the input data. O_5 is the index generated for the graph. O_1, O_3, O_4, and O_6 are vectors used by `bc` algorithm. O_7 is a shared queue among all threads. O_2 is a bitmap to record the successors of each node during the back-propagation phase. The bitmap is shared by all the threads, but since each thread accesses the entries of the bitmap independently, its access exhibits high MLP.

Table 3 shows that under a 50% slow-tier setup, SOAR places the top seven ranked objects (O_1–O_8) in the fast-tier, while the remaining (O_8–O_{10}) are assigned to the slow-tier (note that O_8 spans both tiers). In contrast, "FT" and "Freq" produce different object placement decisions. NoTier allocates O_{10} and part of O_8 to the fast-tier, while the top six most performance-critical objects are placed in the slow-tier. For frequency-based tiering systems, O_1, O_3, O_4, O_5, and part of O_8 are the likely targets for page promotion (per "Freq" rankings in Table 2). Meanwhile, some pages from O_8 are also likely to be selected for demotion. This simultaneous promotion and demotion of O_8 across tiers incurs unnecessary performance overhead due to the lack of object-level performance awareness.

The MLP characteristics of certain objects can mislead frequency-based ranking and cause incorrect page selection, resulting in additional migration overhead. Such effects are ignored by prior tiering designs. In particular, several objects that are ranked low by SOAR but high by "Freq" exhibit high MLP. For example, O_2 ranks 2^{nd} in unit score in SOAR, but is ranked much lower (6^{th}) by "Freq" and "FT" due to low AOL during its active periods (Figure 5). Prior tiering systems will incorrectly prioritize O_3–O_5 and O_8 over truly performance-critical objects like O_2, reducing the likelihood that these more important objects are promoted to the fast-tier.

Takeaway #2: *Performance-aware object placement in SOAR explains its advantage over state-of-the-art tiering designs, whose overhead is exacerbated by ignoring MLP effects.*

6.3 SOAR on CXL

Figure 6 presents the performance of SOAR and baselines on CXL for `bc-urand`, and the trends mirror those observed on the SKX/NUMA setup (Figure 4). SOAR consistently delivers the lowest slowdown across all slow-tier ratios. Nomad shows severe instability with up to 588% slowdown and Colloid experiences up to 92% slowdown, compared to the worst-case slowdown of only 42% in SOAR. Nomad, Colloid, and NBT lose to NoTier almost uniformly across all slow-tier ratios while SOAR is strictly better than NoTier. These results reaffirm that SOAR is effective and robust across both emulated and real CXL environments.

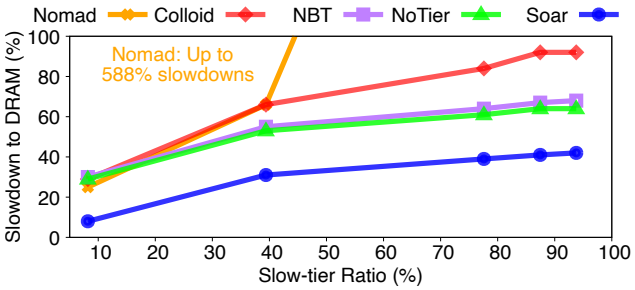


Figure 6: **SOAR vs. others for bc-urand on CXL (§6.3).** *SOAR performance on CXL is similar to that on SKX/NUMA, with SOAR consistently outperforming all baselines across slow-tier ratios.*

Table 4: **SOAR vs. others for more workloads (§6.4).** *SOAR is robust across workloads and consistently delivers better performance compared to existing tiering designs.*

	microbench	bc-urand	bc-twitter	bc-kron	sssp-kron	tc-twitter	603.bwaves
SOAR	34%	16%	7%	18%	14%	7%	4%
Colloid	60%	58%	26%	40%	25%	6%	43%
NBT	58%	68%	13%	59%	18%	11%	13%
Nomad	58%	123%	61%	105%	29%	24%	18%
TPP	58%	875%	495%	792%	760%	38%	1246%
NoTier	46%	67%	63%	55%	39%	9%	9%

6.4 SOAR for More Workloads

Table 4 shows that SOAR consistently achieves the lowest slowdown (up to 18%) across all realistic workloads under a 50% slow-tier ratio. In contrast, state-of-the-art suffer significantly higher slowdowns, with Colloid, NBT, Nomad, TPP, and NoTier reaching up to 58%, 68%, 123%, 1246%, and 67%. "microbench" represents the microbenchmark from §2.1. SOAR outperforms the next best system by 4–42% (except for `tc-twitter`). Even the baseline NoTier outperforms several tiering policies in many cases, highlighting the inefficiencies of page-based migrations. Among all the tiering baselines, no single approach consistently outperforms the others and their performance varies significant across workloads. These results further reinforce the effectiveness of SOAR's performance-aware object allocation across diverse workloads.

Takeaway #3: *SOAR's performance advantage over state-of-the-art and NoTier hold across workloads except for* `tc-twitter` *where SOAR loses by 1% to Colloid.*

6.5 ALTO Performance Evaluation

As the performance of existing tiering designs degrade with increasing slow-tier ratios, we evaluate ALTO with a fast-tier size that is sufficient to accommodate the workload's working set size, as determined by offline profiling, to demonstrate ALTO benefits. This configuration is biased in favor of existing tiering systems. Figure 7a&b compares ALTO with existing tiering policies across a range of workloads on both SKX/NUMA and SPR/CXL. In both environments, ALTO reduces slowdowns

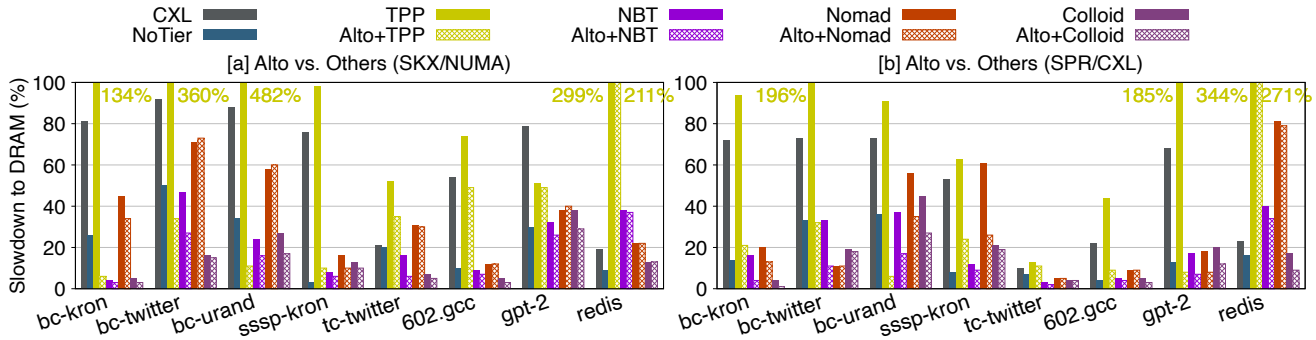


Figure 7: **Alto vs. TPP, NBT, Nomad, Colloid, and NoTier on SKX/NUMA and SPR/CXL across 8 workloads (§6.5–§6.7).** *(a) On SKX/NUMA, Alto+TPP outperforms TPP by 2–471%. Similarly, Alto+NBT outperforms NBT by 1–20%, Alto+Nomad surpasses Nomad by -2–11%, and Alto+Colloid achieves a performance gain over Colloid of 0–9%. (b) On SPR/CXL, Alto exhibits a similar trend to SKX/NUMA, outperforming TPP, NBT, Nomad, and Colloid by 2–178%, 1–23%, 0–35%, and 0–18%, respectively.*

Table 5: **Page promotion reductions of Alto compared to baseline (§6.5).** *Alto significantly reduces the number of page promotions by up to 127.4×.*

	bc-kron	bc-twitter	bc-urand	sssp-kron	tc-twitter	602.gcc	gpt-2	redis
TPP	127.4×	40.0×	83.9×	58.5×	1.5×	2.5×	2.7×	1.8×
NBT	3.5×	1.1×	1.7×	9.4×	1.2×	1.1×	1.9×	1.0×
Nomad	1.2×	2.3×	2.1×	1.4×	1.0×	4.4×	1.3×	1.4×
Colloid	14.9×	1.1×	2.1×	9.2×	1.0×	3.8×	3.7×	1.0×

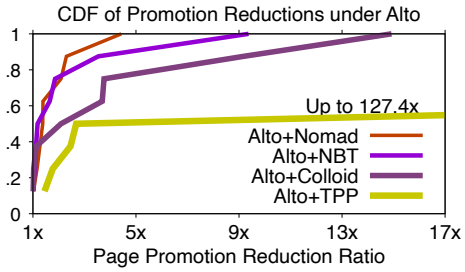


Figure 8: **CDFs of page promotion reductions in Alto (§6.5).** *Alto+Nomad promotion reduction is less than others.*

across all tiering systems (TPP, NBT, Nomad, and Colloid) for majority of the workloads, often cutting performance degradation by more than half. On SKX/NUMA (left), Alto eliminates extreme outliers, resulting in significantly better performance of only 2% slowdown in the best case. On CXL (right), the overall performance trend remains similar: Alto-enhanced policies outperform their baselines across majority of the workloads, highlighting Alto's relative robustness across different system configurations.

Among the four baselines, TPP exhibits the most aggressive migration behavior due to its intensive hot-page detection strategy. As a result, it performs poorly in our evaluation, reaching up to 482% slowdown, which is even worse than running the workloads entirely on the slow-tier ("CXL" bars).

Colloid focuses on balancing fast- and slow-tier latency under bandwidth saturation, based on the observation that fast-tier latency can exceed that of slow-tier in such cases. However, in our setup, none of the workloads saturate fast-tier bandwidth. By design, Colloid adopts a more aggressive page promotion policy than NBT, promoting hot pages not only to the fast-tier, but also to the slow-tier proactively. While this aggressiveness benefits certain workloads, it is not universally effective. As shown in Figure 7, Colloid performs worse than NBT on nearly half of the workloads.

Nomad adopts a non-exclusive page migration strategy, retaining pages in both tiers to avoid blocking migrations. As a result, reducing migration frequency has a more limited impact on performance compared to other systems. Additionally, Alto+Nomad regulates fewer page promotions than Alto with TPP, NBT, and Colloid, as shown in Table 5. These factors help explain the limited performance improvements of Alto+Nomad observed in several workloads. Alto+Nomad can underperform Nomad by up to 2% on several workloads, such as `bc-twitter`, `bc-urand`, and `gpt-2`. The underlying reasons are not yet clear to us and a deeper analysis is needed to understand the root causes as future work.

Alto's adaptive page promotion regulation allows it to filter out unnecessary page migrations, thus achieving better performance than the corresponding baselines. Table 5 and Figure 8 shows the reduction in page promotions achieved by Alto compared to the corresponding baselines across all eight workloads. Alto significantly reduces the number of page migrations by up to 127.4× while maintaining superior performance for most of the cases.

Takeaway #4: *Alto outperforms TPP, NBT, Nomad, and Colloid by 2–471%, 1–23%, -2–35%, and 0–18% across 8 workloads on NUMA and CXL by regulating unnecessary page promotions effectively. Alto+Nomad loses to Nomad for a few workloads by no more than 2%.*

6.6 Understanding Alto Performance

Alto's strength lies in the simplicity and accuracy of its AOL-based predictor, enabling it to significantly reduce page promotion overhead while improving performance. Figure

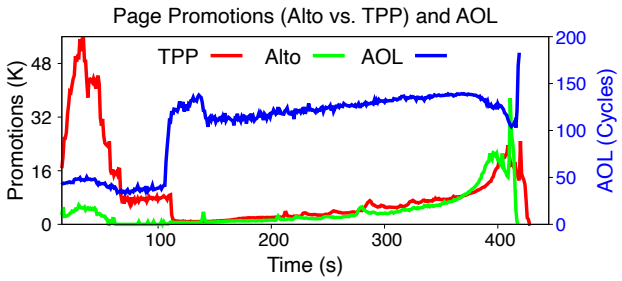


Figure 9: **Page promotions in Alto vs. TPP over time (§6.5–§6.6).** *Alto filters out unnecessary page migrations based on runtime AOL dynamically.*

8 shows the CDFs of reductions in page promotions for Alto compared to the baselines. Specifically, Alto reduces promotions by up to 127.4×, 9.4×, 4.4×, and 14.9× for TPP, NBT, Nomad, and Colloid, respectively.

To better understand this improvement, we zoom in on the `tc-twitter` workload and examine how Alto adaptively regulates promotion behavior. Figure 9 plots the page promotion rate and AOL over time for both TPP and Alto+TPP. Two key observations highlight Alto's effectiveness:

(1) Unlike TPP, which aggressively promotes pages in the first 100 seconds (red line) due to high LLC misses, Alto promotes far fewer pages (green line) during this phase. This is because AOL remains low (blue line), indicating high MLP and limited performance sensitivity to memory latency. By avoiding unnecessary migrations, Alto reduces the number of promoted pages from 1.6 million (in TPP) to just 190K, a reduction of 8.4×.

(2) As the workload progresses, Alto gradually increases its promotion rate in response to rising AOL values, whereas TPP simply follows access intensity without adaptive control. Ultimately, Alto outperforms TPP while migrating 3.5× fewer pages overall.

Other workloads in Figure 7 exhibit similar trends, demonstrating Alto's ability to maintain performance while minimizing migration cost through adaptive, AOL-guided tiering.

6.7 Alto on CXL

Figure 7b details Alto's performance on CXL, which closely mirrors the trends observed on the SKX/NUMA setup (Figure 7a). Across all workloads, Alto improves performance when layered over existing tiering systems, often achieving the lowest slowdown. The performance gains are even more stable on CXL due to improved bandwidth compared to the SKX/NUMA setup. These results highlight Alto's robustness and portability across different hardware configurations. Alto's performance benefits are similarly explanined as Figure 8. Note that while workloads run faster on SPR compared to SKX as the recent SPR CPU is more performant, the slowdown relative to DRAM performance is comparable, or slightly worse, due to the larger fast-tier/slow-tier latency gap (2.4× on SPR vs. 2.1× on SKX).

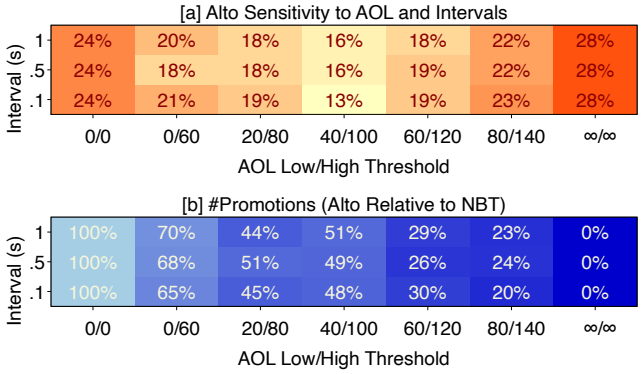


Figure 10: **Alto sensitivity to AOL and sampling intervals (§6.8).** *(a) Alto+NBT performance under varying AOL and intervals. (b) Relative number of promotions in Alto+NBT to NBT.*

6.8 Sensitivity Study

In this section, we study Soar and Alto's sensitivity to AOL and other parameters as well as justifying their default choices.

Soar. Algorithm 1 uses $R_{\min}$, $R_{\max}$, and L_0 as thresholds to incorporate MLP effects into object scoring. We evaluate the sensitivity of Soar to these parameters. Varying L_0 between 70, 80, 90, and 100 cycles results in minimal changes to the object ranking: only the ordering of O_2–O_4 in Table 2 differs. For a 50% slow-tier ratio in `bc-urand`, objects O_1–O_6 and part of O_7 are always placed in fast-tier, yielding identical performance across the full range of L_0 values. This indicates that the maximum AOL threshold in Algorithm 1 is robust.

Similarly, adjusting $R_{\min}$ and $R_{\max}$ to $(0.02, 0.6)$, $(0.03, 0.7)$, and $(0.04, 0.8)$ results in minor changes, only the ranking of O_2–O_5 differs. Since the top objects remain unchanged (under the same slow-tier ratio), the object placement and performance are unaffected, confirming that Soar's scoring is stable under reasonable threshold variations.

Takeaway #5: *Soar is robust to L_0 (from 70 to 100), $R_{\min}$, and $R_{\max}$ (from $(0.02, 0.6)$ to $(0.04, 0.8)$).*

Alto. We evaluate Alto's sensitivity to the AOL thresholds ($\texttt{AOL}_{\texttt{low}}$, $\texttt{AOL}_{\texttt{high}}$), and AOL sampling periods. Figure 10 shows the Alto results under a range of ($\texttt{AOL}_{\texttt{low}}$, $\texttt{AOL}_{\texttt{high}}$) threshold pairs and sampling intervals (100ms, 500ms, 1s).

In Figure 10a, we observe that Alto maintains stable performance across a wide range of threshold values. Slowdown remains within 16–22% for most parameter combinations, indicating robustness to threshold tuning. Extreme thresholds (*e.g.*, $0/\infty$) lead to degraded performance, highlighting the importance of filtering based on AOL to avoid over- or under-migration. Figure 10b shows the relative number of page promotions compared to Colloid. Across most configurations, Alto reduces promotions by 30–70%, while preserving performance. With more conservative AOL thresholds (*e.g.*, 60/120) and shorter sampling intervals, Alto eliminates majority of the page promotions (down to 30%) while still maintaining

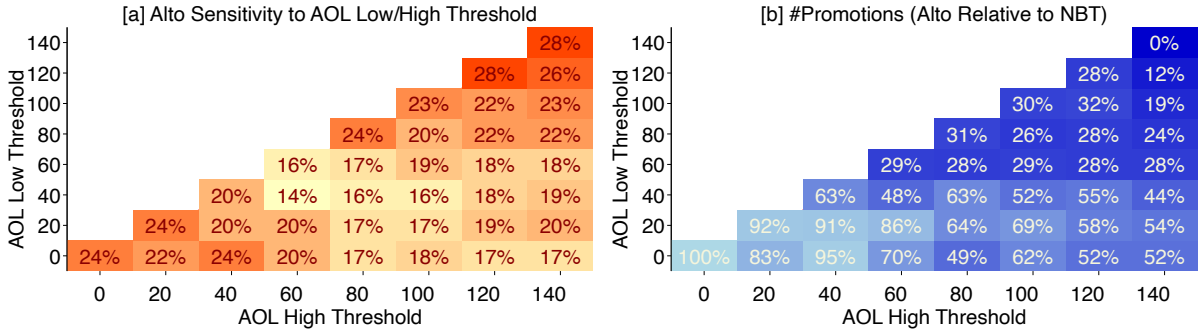


Figure 11: **Alto sensitivity to more detailed AOL low/high thresholds (§6.8).** *(a) Alto's performance can vary by up to 14% across different combinations of AOL low/high thresholds, and its default threshold setting is close to the optimal configuration. (b) Alto reduces page promotions according to AOL low/high thresholds.*

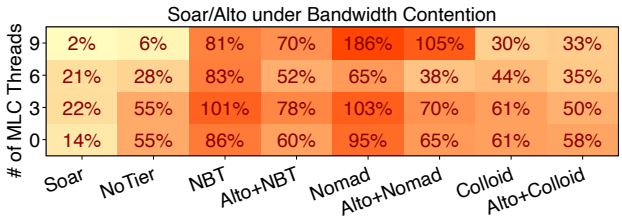


Figure 12: **Soar and Alto performance slowdown under bandwidth contention (§6.9–§6.10).** *Under bandwidth contention, Soar continues to outperform both NoTier and other tiering policies, and Alto+NBT, Alto+Nomad, and Alto+Colloid improve over their baselines by 11–26%, 27–81%, and -3–11%, respectively.*

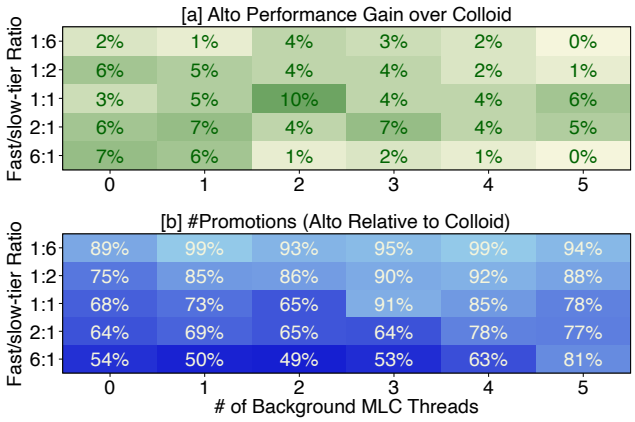


Figure 13: **Alto+Colloid performance gains over Colloid under bandwidth contention and different fast/slow-tier ratios (§6.10).** *Alto+Colloid outperforms Colloid by up to 10% under most fast/slow-tier ratios combined with bandwidth contention.*

acceptable performance (19% vs. 16% slowdown).

Figure 11 shows the sensitivity of Alto to varying AOL low/high threshold values, with combinations ranging from 0 to 140 (observed highest AOL). The two extreme cases (0/0 and 140/140) in Figure 11 represent no promotion regulation (*i.e.*, equivalent to the NBT baseline) and full regulation (*i.e.*, equivalent to NoTier), respectively. Across the threshold range from bottom left to top/bottom right, Alto's performance typically correlates with the degree of regulation: more page promotion regulation (smaller numbers in Figure 11b) generally leads to better performance (smaller slowdowns in Figure 11a). The default setting used by Alto (40/100 on SKX/NUMA) delivers reasonably good, though not optimal, performance (16% vs. 14% slowdown). We observe a maximum performance difference of 14% between 40/60 and 140/140 (NoTier). While the number of regulated promotions generally reflects Alto's efficiency, the exact performance impact of each regulation is hard to isolate and needs further investigation.

Takeaway #6: *Alto is robust to sampling intervals from 100ms to 1s, and its default AOL low/high thresholds are near-optimal, thanks to the accurate modeling of K and AOL.*

6.9 Soar under Bandwidth Contention

To evaluate the performance robustness of Soar to bandwidth pressure, we use Intel MLC [55] to generate memory traffic on the fast-tier by varying the number of threads from 0 to 9 on SKX. Each MLC thread sustains ~8 GB/s. The remaining 1 core is reserved for `bc-urand` with 50% slow-tier ratio. With 9 MLC threads, the bandwidth reaches 48 GB/s, which is 98% of the total bandwidth. Latency increases to 180 ns, 2× of the unloaded latency, demonstrating significant queuing delays.

Figure 12 shows that Soar outperforms the second best by 4–41% under bandwidth contention. As contention increases (0–9 MLC threads), Colloid consistently outperforms Nomad and NBT, but underperforms NoTier, with the performance gap widening under higher contention. This indicates that tiering becomes less effective in bandwidth-bound scenarios. Soar maintains its performance lead over NoTier (the second-best), though its gains diminish as contention intensifies (33% vs. 4% for 3 and 9 MLC threads).

Takeaway #7: *State-of-the-art tiering designs consistently underperform NoTier due to elevated migration overhead under bandwidth contention. In contrast, Soar outperforms all of them on bc-urand, though its performance gains decrease as contention increases, reaching up to 2× inflated latency.*

6.10 Alto under Bandwidth Contention

Figure 13a further compares Alto+Colloid and Colloid under varying fast/slow-tier ratios (ranging from 1:6 to 6:1) in a different MLC setup: 5 cores are reserved for `bc-urand`, and the remaining 5 cores run MLC threads. With 5 MLC threads, the system bandwidth reaches 40GB/s, approximately 81% of the total bandwidth, and latency increases to 140ns, which is 56% higher than the unloaded latency of 90ns.

Under moderate memory bandwidth contention in this case, even though overall system MLP increases, the background memory pressure does not alter the foreground workload's MLP behavior. While contention raises memory latency, it does not reduce the workload's inherent MLP (confirmed via our MLP measurements). However, as MLC threads increase from 0 to 5, the AOL range shifts from 30–140 to 40–250. Alto remains effective at regulating page promotions, as only 14.2% of the runtime experiences AOL above 100. Figure 13a&b supports this claim, showing that Alto+Colloid outperforms Colloid under bandwidth contention while migrating up to 51% fewer pages. While contention narrows the latency gap between tiers (*i.e.*, fast-tier latency approaches that of the slow-tier), the MLP-induced performance penalties of page promotion persist.

When further increasing the number of MLC threads to 9, Figure 12 shows somewhat diminished Alto performance gains over its baseline, in particular, Alto+Colloid vs. Colloid, with -3–81% improvement. Among all the baselines, Nomad shows the worst performance under extreme bandwidth contention, while Colloid performs the best (its target scenario). Note that `bc-urand` uses only 1 core here, as more cores are reserved for MLC, compared to 5 cores in Figure 13. Alto+Colloid falls behind Colloid by 3% under the 9-MLC-thread setup but outperforms it in all other cases. This behavior stems from increased fast-tier latency under bandwidth contention, which expands the AOL range from 40–140 (0 MLC threads) to 95–270 (9 MLC threads). With default AOL thresholds (40/100), Alto's regulation becomes less aggressive (Algorithm 2) when AOL falls out of the target AOL range, as shown in Figure 13b. Only 2.3% of runtime phases exhibit AOL below 95. As a result, Alto regulates only 1.6% of pages under 9 MLC threads, leading to 3% negative performance gains. However, adjusting AOL thresholds to 90/150 and 90/270 to match the runtime AOL range improves Alto+Colloid's slowdown from 33% to 23% and 20%, both outperforming Colloid (30%). Thus, adjusting AOL thresholds upward to match runtime AOL under high bandwidth contention can help preserve Alto's benefits. This requires tuning AOL thresholds based on contention levels. Developing an auto-tuning mechanism for AOL thresholds for such scenarios is an interesting direction for future work.

Takeaway #8: *Alto achieves improvements of -3% to 81% over Nomad, NBT, and Colloid for bc-urand under bandwidth pressure across various fast/slow-tier ratios. However, Alto's benefits diminish under extreme contention (e.g., 2× inflated latency), resulting in one case where Alto+Colloid underperforms Colloid by 3%. Raising AOL thresholds can help recover performance benefits in such scenarios.*

7 Related Work

Heterogeneous Memory Management. Soar shares its profiling-based design goal with prior systems [13, 34, 49, 56], but differs in its use of performance metrics, leading to different design choices. Unlike X-Mem [13], which classifies memory access to static types of patterns and ranks coarse-grained memory regions using offline-profiled latency, Soar leverages an AOL-based predictor that accounts for MLP and latency inflation for accurate slowdown prediction (§3.2). Soar is application-transparent without code changes and is lightweight, imposing neglible runtime overhead, making it applicable to a wide range of workloads.

Memory Performance Modeling. Prior work has focused on modeling memory performance using various metrics, including CPU stalls, LLC misses, memory latencies, etc., as well as ML-driven predictors [7, 8, 11, 21, 23, 57]. However, these approaches often fall short due to accuracy and complexity issues. While CPU stalls on the fast-tier can intrinsically capture memory access performance impact, they serve as a poor metric to predict slow-tier performance due to their inability to account for shifting memory-overlap effects caused by increased latency and variable MLP in the slow-tier. In contrast, our AOL-based predictor explicitly models CPU stalls, latency, and MLP. AOL amortizes the overestimated CPU stall increases, ensuring high prediction accuracy and serving as the key to Soar/Alto's effectiveness.

Memory Tiering. Memory tiering has been extensively studied from many angles, including efficient software- and hardware-based hotness tracking, memory allocation and migration policies across host and virtualized environments, and support for various slow-tier memory types [4, 13–28, 58, 59]. For instance, HeMem utilizes PEBS for fine-grained page access frequency sampling [17]. Most existing tiering systems rely on hotness for guiding data placement across tiers, ignoring the performance impact variability across memory accesses. Soar and Alto designs are orthogonal to existing tiering designs, acting as a general-purpose memory allocator and migration regulator that complements other optimizations such as better hotness tracking and migration policies.

8 Conclusion

Tiered memory management is becoming increasingly important with the rise of CXL, yet significant challenges persist. We demonstrate that hotness does not equate to performance, highlighting the need to revisit both the fundamental principles and strategies for memory tiering. We hope that our predictive metrics on AOL, static and dynamic tiering policies (Soar and Alto) will open up new directions for memory research.

Acknowledgments

We thank Philip Levis, our shepherd, and the anonymous OSDI'25 reviewers for their constructive feedback, which significantly improved this paper. We also thank CloudLab for providing the infrastructure. This research was partially supported by an NSF CAREER Award (CNS-2339901), NSF Grant CNS-2312785, Samsung, Google, and Microsoft.

References

[1] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[2] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[3] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[4] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[5] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. Design Tradeoffs in CXL-Based Memory Pools for Cloud Platforms. *IEEE Micro Special Issue on Emerging System Interconnects*, 43(2), 2023.

[6] Jacob Wahlgren, Gabin Schieffer, Maya Gokhale, and Ivy Peng. A Quantitative Approach for Adopting Disaggregated Memory in HPC Systems. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2023.

[7] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[8] Yan Sun, Yifan Yuan, Zeduo Yu, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-56)*, 2023.

[9] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the 2024 EuroSys Conference (EuroSys)*, 2024.

[10] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.*, 56(11), July 2024.

[11] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[12] Xi (Sherry) Wang, Jie Liu, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. Performance Characterization of CXL Memory and Its Use Cases. In *Proceedings of the 39th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.

[13] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the 2016 EuroSys Conference (EuroSys)*, 2016.

[14] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent Page Management for Two-

tiered Main Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[15] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenters. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[16] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[17] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[18] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *Proceedings of the 28th International Symposium on High Performance Computer Architecture (HPCA-28)*, 2022.

[19] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[20] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the 2024 EuroSys Conference (EuroSys)*, 2024.

[21] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[22] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. NOMAD: Non-Exclusive Memory Tiering via Transactional Page Migration. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[23] Midhul Vuppalapati and Rachit Agarwal. Tiered Memory Management: Access Latency is the Key! In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[24] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Jie Zhang, Yongqiang Xiong, and Guangyu Sun. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering. In *57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-57)*, 2024.

[25] Dong Xu, Junhee Ryu, Jinho Baek, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*, 2024.

[26] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. Telescope: Telemetry for Gargantuan Memory Footprint Applications. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*, 2024.

[27] Zhenlin Qi, Shengan Zheng, Ying Huang, Yifeng Hui, Bowen Zhang, Linpeng Huang, and Hong Mei. Chrono: Meticulous Hotness Measurement and Flexible Page Migration for Memory Tiering. In *Proceedings of the 2025 EuroSys Conference (EuroSys)*, 2025.

[28] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[29] Musa Unal, Vishal Gupta, Yueyang Pan, Yujie Ren, and Sanidhya Kashyap. Tolerate It if You Cannot Reduce It: Handling Latency in Tiered Memory. In *Proceedings of the 20th Workshop on Hot Topics in Operating Systems (HotOS XX)*, 2025.

[30] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[31] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[32] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair

Memory Access Scheduling for Chip Multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, 2007.

[33] Aditya Narayan, Tiansheng Zhang, Shaizeen Aga, Satish Narayanasamy, and Ayse Coskun. MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[34] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[35] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[36] Memory Tiering: Hot Page Selection. https://lwn.net/Articles/898615/.

[37] mm/demotion: Memory Tiers and Demotion. https://lwn.net/Articles/897026/.

[38] Better Support for Locally-attached-memory Tiering. https://lwn.net/Articles/974126/.

[39] Latency Oriented Processor Architecture. https://en.wikipedia.org/wiki/Latency_oriented_processor_architecture.

[40] Adar Zeitak and Adam Morrison. Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[41] SPEC CPU 2017. https://www.spec.org/cpu2017.

[42] GAP Benchmark Suite. https://github.com/sbeamer/gapbs.git, 2024.

[43] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization. https://arxiv.org/abs/2409.14317, 2024.

[44] Michaelis-Menten Kinetics. https://en.wikipedia.org/wiki/Michaelis-Menten_kinetics, 2024.

[45] NUMA Memory Policy. https://docs.kernel.org/admin-guide/mm/numa_memory_policy.html.

[46] Memkind. https://github.com/memkind/memkind, 2024.

[47] Unified Memory Framework. https://github.com/oneapi-src/unified-memory-framework, 2024.

[48] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A Progam-Behavior-Guided Far Memory System. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[49] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[50] Sotiris Apostolakis, Chris Kennelly, Xinliang David Li, and Parthasarathy Ranganathan. Necro-reaper: Pruning away Dead Memory Traffic in Warehouse-Scale Computers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[51] Huang Ying. AutoNUMA: Optimize Memory Placement for Memory Tiering System. https://lwn.net/Articles/835402/.

[52] The CloudLab Manual - Hardware. https://docs.cloudlab.us/hardware.html, 2025.

[53] GPT-2. https://en.wikipedia.org/wiki/GPT-2.

[54] Redis. https://redis.io.

[55] Intel Memory Latency Checker (Intel MLC). https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html.

[56] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012.

[57] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th International Middleware Conference (Middleware)*, 2015.

[58] Samir Rajadnya and Durgesh Srivastava. CMS: Hotness Tracking Requirements. https://www.opencompute.org/documents/ocp-cms-hotness-tracking-requirements-white-paper-pdf-1.

[59] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-Based Hybrid Memory Management. In *International Conference on Cluster Computing (Cluster)*, 2017.

NanoFlow: Towards Optimal Large Language Model Serving Throughput

Kan Zhu
University of Washington

Yufei Gao
University of Washington
Tsinghua University

Yilong Zhao
University of Washington
UC Berkeley

Liangyu Zhao
University of Washington

Gefei Zuo
University of Michigan

Yile Gu
University of Washington

Dedong Xie
University of Washington

Tian Tang
University of Washington
Tsinghua University

Qinyu Xu
University of Washington
Tsinghua University

Zihao Ye
University of Washington

Keisuke Kamahori
University of Washington

Chien-Yu Lin
University of Washington

Ziren Wang
University of Washington
Tsinghua University

Stephanie Wang
University of Washington

Arvind Krishnamurthy
University of Washington

Baris Kasikci
University of Washington

Abstract

Large Language Models (LLMs) have resulted in a surging demand for planet-scale serving systems, where tens of thousands of GPUs continuously serve hundreds of millions of users. Consequently, throughput has emerged as a key metric that determines serving systems' performance. Due to large model sizes and memory-intensive self-attention, LLM serving has been commonly assumed to be memory-bound. Through a detailed analysis, we show that despite having memory-intensive components, end-to-end LLM serving is compute bound for most common workloads and LLMs. Alas, most existing serving engines fall short from optimal compute utilization, because the heterogeneous operations that comprise LLM serving—compute, memory, networking—are executed sequentially within a device.

We propose NanoFlow, a novel serving framework that exploits intra-device parallelism, which overlaps the usage of heterogeneous resources within a single device. NanoFlow splits inputs into smaller *nano-batches* and duplicates operations to operate on each portion independently, enabling overlapping. NanoFlow automatically identifies the number, size, ordering, and GPU resource allocation of nano-batches to minimize the execution time, while considering the interference of concurrent operations. We evaluate NanoFlow's end-to-end serving throughput on several popular models such as LLaMA-2-70B, Mixtral 8×7B, LLaMA-3-8B, etc. With practical workloads, NanoFlow provides 1.91× throughput boost compared to state-of-the-art serving systems, achieving between 50% to 72% of optimal throughput across popular models.

1 Introduction

Large language models (LLMs) based on transformers power applications like chatbots, search engines, and office software but are highly resource-intensive [21, 28, 41]. Reports indicate over 200 million weekly ChatGPT users, with API usage doubling after GPT-4o Mini's release [31, 35, 40]. Given the massive demand and limited GPU availability [11, 47], maximizing hardware utilization is critical. In particular, throughput, measured as tokens per device per second, has become a key factor in reducing serving costs [13, 17, 26].

LLMs are more resource-intensive than earlier DNNs due to unique characteristics. Their model sizes are vastly larger, for example, GPT-3 with 175B parameters requires five A100 80 GB GPUs to store weights in 16-bit precision, with some closed-source models even larger [7]. Moreover, LLMs' self-attention mechanism [51] scales total memory loading quadratically with the context length—the aggregate length of input and output sequences—during token generation. LLM serving systems often cache per-request state in a *key-value cache* (KV-cache), whose size can surpass the size of model weights [44], driving up memory demands. Moreover, each LLM iteration loads the entire model weights plus the KV-cache, while only producing one output token for each

decoding sequence, further increasing memory pressure. Due to these characteristics, LLM serving is commonly assumed to be overall memory bandwidth bound.

In this paper, we show that while the self-attention operation is indeed memory-bound, *LLM serving is often compute-bound when considered as a whole*. We provide a detailed analysis (which we also validate empirically) of the common operations used in current LLMs and find that: (1) Batching prefill and decode requests amortizes weight loading, making general matrix multiplications (GEMMs) compute-bound. (2) Although batched decode requests still need to load a unique KV-cache per sequence, novel optimizations such as grouped query attention [5] (GQA) can reduce the memory loads. (3) As model sizes grow, the compute operations from GEMMs tend to dominate compared to self-attention. Consequently, for many common workloads and LLMs, total compute operations dominate network and memory I/O.

However, in practice, we find that LLM serving engines are far from optimal in terms of compute utilization when measured end-to-end and compared to the hardware FLOPs. This is because LLMs use *heterogeneous operations*. For example, the self-attention is memory-bound during the decode phase. Also, when models are larger than a single GPU's memory capacity, GEMM is often split across multiple GPUs when using tensor parallelism, and thus requires network communication in addition to compute. We find that in current LLM serving engines, individual operations have high utilization of their bottlenecked resource ($\sim$ 80%). However, because each operation is executed one at a time, and different operations have different resource bottlenecks, the total compute utilization is only 40%, as shown in Section 6.2.

To address this gap, we present NanoFlow, a serving framework that aims to maximize utilization of the workload's resource bottleneck, when considered as a whole. The key insight is to leverage *intra-device parallelism for heterogeneous operations*. In particular, NanoFlow splits each input batch into *nano-batches* and duplicates operations across nano-operations, with each nano-operation processing a single nano-batch. Since nano-operations operate on separate nano-batches without dependencies, heterogeneous operations—such as memory-bound and compute-bound operations—can execute *simultaneously* rather than sequentially. This approach facilitates fine-grained pipelining across compute, memory, and network resources within a *single device*. As nano-operations duplicate each operation, this method increases memory I/O for weight loading. However, when the workload is compute-bound as a whole, the additional memory I/O can be hidden via pipelining.

To achieve efficient pipelining, NanoFlow executes nano-operations with a fraction of the GPU's resources. For instance, while most of the GPU resources can be assigned to a compute-heavy operation for high compute utilization, a memory-bound operation can still achieve considerable memory bandwidth usage using the remaining GPU resources.

Designing NanoFlow presents several major challenges. First, given the multitude and diversity of LLMs, determining the appropriate number of nano-batches, the input batch size for each nano-operation, and the execution order of nano-operations involves navigating a large search space. Second, when memory- or network-bound kernels run alongside compute-bound kernels, competition for GPU resources (e.g., execution units, caches) causes unpredictable performance interference, slowing down kernels running in parallel [43]. Balancing these trade-offs is critical. Allocating more resources to compute-bound kernels may excessively delay memory- or network-bound kernels, while prioritizing memory- or network-bound kernels could hinder overall throughput by slowing down compute-bound operations.

To tackle these challenges and efficiently explore the large search space, NanoFlow proposes *auto-search* to automatically construct an intra-device pipeline. The auto-search process reduces the search space by executing in two stages, which approximates the optimal solution balancing efficiency and accuracy. In the first stage, NanoFlow determines the initial pipeline schedule, including the number, size, and ordering of nano-operations, assuming no interference between kernels. In the second stage, NanoFlow refines the pipeline by profiling actual kernel interference and re-planning accordingly. NanoFlow also provides a runtime system to execute the optimized pipeline by efficiently forming nano-batches, allocating GPU resources, and managing the KV-cache.

We provide a detailed evaluation of NanoFlow on the LLaMA-2-70B model [49] using one NVIDIA 8×A100 DGX node [24]. For practical workloads including ShareGPT [1], LMSys [56] and Splitwise [32], NanoFlow achieves on average 1.91× greater throughput than state-of-the-art serving frameworks including vLLM [17], DeepSpeed-FastGen [13] and TensorRT-LLM [26], which is 68.5% of the theoretically optimal throughput we derive in §3. NanoFlow achieves similar latency compared with the best baseline TensorRT-LLM at a low request rate, while handling up to 1.64× higher request rates within SLO constraints. We also demonstrate NanoFlow's ability to automatically generate efficient pipelines for other models and architectures such as Mixture-of-Experts (MoE), by evaluating popular LLMs (LLaMA-3-70B, LLaMA-3-8B, QWen2-72B, Deepseek-67B, and Mixtral 8×7B). NanoFlow-generated serving pipelines achieve 50%-72% of optimal throughput, which on average reaches 2.66× throughput gain compared with vLLM.

In summary, we contribute the following:

- A detailed analysis and empirical validation of the workload characteristics and theoretically optimal throughput of LLM serving systems that shows modern LLMs operate in a compute-bound regime.
- NanoFlow, that comprises (1) an auto-search engine that automatically computes a pipeline for nano-batches to improve LLM serving throughput, (2) an end-to-end LLM serving runtime.

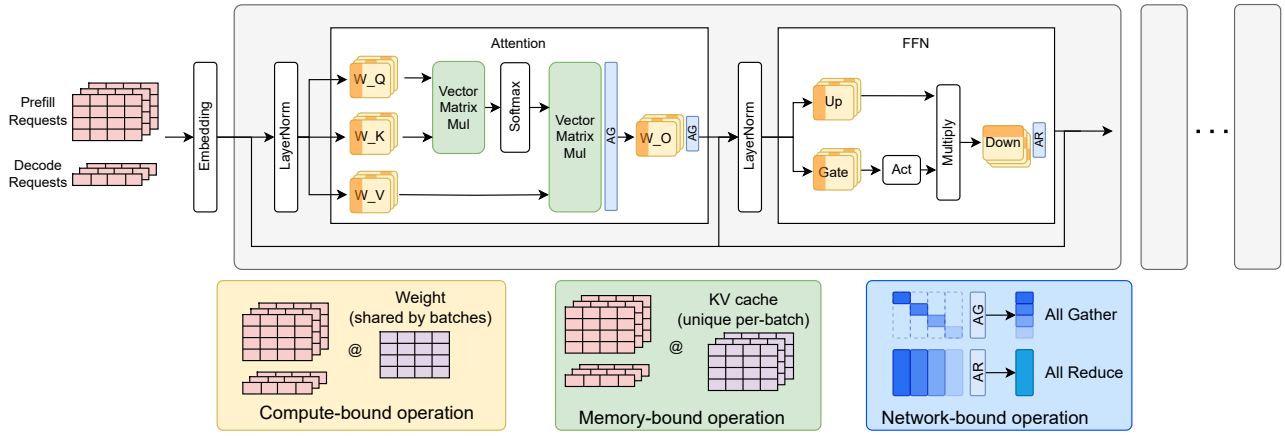


Figure 1: Transformer architecture. The operations in the yellow boxes have large batch sizes and share model weight parameters across requests; hence, they are compute-bound. Operations in green boxes require loading a unique KV cache for each request; hence, they are memory-bound. The blue box represents network operations that perform synchronization between operations.

- A comprehensive evaluation of NanoFlow that achieves 1.91× throughput gain compared to baselines and 68.5% of theoretical maximum throughput.

2 Background

2.1 LLM Inference Workflow

Recent LLMs like GPT-4, LLaMA, and Mistral [16, 29, 49] are based on the decoder-only transformer [51]. The inference process of these LLMs has two phases [53]: (1) *prefill*, which processes the input prompt all at once, and (2) *decode*, which generates output tokens one at a time autoregressively. The prefill phase initializes the *KV-cache* [34], which keeps the per-request state to speed up the decode phase.

Both the prefill and decode phases utilize the same inference workflow in Figure 1. Upon receiving an inference request, the input traverses identical *transformer layers* to produce the subsequent tokens. In each transformer layer, during the attention stage, the input is multiplied by weight matrices W_Q, W_K, and W_V to form Query, Key, and Value, with Key and Value concatenated into the existing KV-cache. The Query is then multiplied with the existing Keys and normalized to assess the similarity between the current token and all previous tokens. This similarity is employed to compute the weighted average of the Value, aggregating the context information. The outcome undergoes O-projection through a linear transformation using W_O followed by a layer normalization operation [6]. Subsequently, during the feed-forward network stage, the activation is multiplied by W_{up} and W_{gate} separately to generate o_u and o_g. Next, o_g passes through an activation function (e.g., SiLU [10]), followed by an element-wise multiplication with o_u. Finally, W_{down} is applied as the Down-projection to generate the output, which serves as the input to the next layer.

2.2 Operation Characteristics

According to the characteristics of the operations during inference, we can classify them into four categories.

Dense operations. The KQV generation, O projection, Up, Gate, and Down projection compute the multiplication of activations and model weights, which we define as *dense operations*. In the prefill phase, all tokens of new requests form a large batch for these dense operations. In the decode phase, batched decoding aggregates the activations from all the decode requests for dense operations [53]. Moreover, as the prefill phase and decode phase share weight matrices, the activations from both phases can be combined to further exploit the batching effect by amortizing the weight loading cost [3]. As a result, dense operations are *compute-bound*.

Attention operations. Transformers capture token relations through attention [51]. The prefill phase processes all input tokens simultaneously, making it *compute-bound*, while the decode phase generates one token per iteration, loading cached K and V vectors from the KV-cache, making it *memory-bound*.

Network operations. Collective communication operations including AllGather (AG) and AllReduce (AR) are *network-bound* and call for high bandwidth interconnection between nodes like NVLink [25].

Other operations. We classify the rest of the operations in transformer architecture, e.g., layer norms, positional embeddings, etc., as other operations, due to their short runtime compared to dense, attention or network operations.

2.3 Serving LLMs at Scale

With large models, multiple GPUs are required to provide enough resources, including both memory and compute, for efficient LLM serving [18]. Existing works [12, 38, 57] have comprehensive evaluations of different *inter-device paral-*

lelism paradigms:

Tensor Parallelism [38, 57] splits each weight matrix onto different GPUs and executes each operation collectively, avoiding weight duplication, thereby scaling compute capacity. However, frequent collective communications after operations are required to synchronize each operation's results, as shown in Figure 1.

Pipeline Parallelism [14] splits the model into stages by layers so that each GPU only needs to hold part of the weights. While all devices with tensor parallelism execute on the same batch of data, devices with pipeline parallelism operate on different micro-batches of data split by stages.

In practice, these parallelism paradigms are typically used in combination. Tensor parallelism is commonly applied across GPUs within a single node, where high-bandwidth interconnects are available, to support larger batch sizes. Pipeline parallelism, on the other hand, is employed across nodes to further scale the system.

3 Analysis

We now introduce the factors determining the throughput of LLM serving (§3.1) and model the cost of LLM serving (§3.2). We then classify modern LLM serving workloads according to our model (§3.3) and empirically validate this classification (§3.4), showing that popular workloads are compute-bound. Then, for the compute-bound scenarios, we determine optimal serving throughput (§3.5), explain why existing systems fall short of the optimal (§3.6), and how this motivates our proposed approach, intra-device parallelism (§3.7).

3.1 Key Factors of Serving Throughput

We define throughput as the *total throughput* of serving, which is the number of tokens processed per second by both prefill and decode phases. Throughput is affected by hardware properties, model configuration, user queries, and batch size.

Hardware specification. The following factors determine the available GPU hardware resources:

- N_{GPU}: number of GPUs
- $MemBW$ (GB/s): aggregate GPU memory bandwidth
- $MemSize$ (GB): aggregate GPU memory capacity
- $Compute$ (GFLOP/s): aggregate GPU compute capacity
- $NetBW$ (GB/s): aggregate GPU interconnect bandwidth

Model configuration. The following factors regarding the model architecture determine the computation, memory, and network demand when serving LLMs:

- D_{model}: hidden dimension size
- L: number of layers
- P_{Model}: number of parameters
- R_{GQA}: group size of GQA [1]
- S_{type} (Bytes): number of bytes (size) of the data type for model parameters (e.g., 2 for FP16)

User query statistics. The following user input statistics also affect the system throughput:

- p: average number of tokens in prompts to be prefilled
- d: average number of tokens in output to be decoded

Therefore, a serving request from a user corresponds to p prefill tokens and d decode tokens on average, for a total of $p+d$ tokens that we use when calculating the total throughput. Using user query statistics, total throughput (tokens generated per second) can be easily converted to other throughput metrics, for example, decoding throughput (decode output per second) is $\frac{d}{p+d} \times \text{Throughput}_{\text{total}}$, and request per second is $\frac{1}{p+d} \times \text{Throughput}_{\text{total}}$.

Batch size. An optimal throughput-oriented serving system needs to process large batches. For compute-bound dense operations, e.g., GEMMs, a larger batch amortizes the weight loading overhead and increases execution unit occupancy, which provides higher compute utilization. For memory- and network-bound operations, all requests share the same warmup cost (e.g., kernel launching overhead, pipeline setup overhead, network synchronization overhead, etc.), so a larger batch size leads to higher throughput. Therefore, when considering the optimal system throughput, we assume that the system always operates at the largest batch size at which the total available memory can hold the model weights and all the KV caches for the requests. [2]

3.2 Cost Model of LLM Serving

Under the assumption of maximum batch size, we derive the latency of an LLM serving iteration (where a batch of user requests are being processed by all the transformer layers) from the perspectives of required memory, compute, and network resources.

Memory. From the memory perspective, latency is:

$$T_{mem} = \frac{MemSize}{MemBW} \tag{1}$$

This is because the entire device memory content needs to be loaded into the GPU caches and registers once for each iteration (typical in modern serving systems [17]) when using the largest batch size. Although the model weights are shared between iterations, due to the long reuse distance, it is infeasible to cache the model weights and avoid loading from the device memory.

Compute. In LLM serving, all operations contribute to the total compute, while Dense operations produce the vast majority of computations for typical workloads (as we validate in §3.4). Therefore, we model the compute latency of dense operations below.

For every GEMM in dense operations, $2B_{Dense}N_wK_w$ computations need to be performed on weights and activations, where N_w and K_w are dimensions of the weight matrices, and B_{Dense} is the batch size for dense operations, which includes decode tokens from hundreds of requests and the prefill tokens from one or a few requests. Thus, the total compute for

[1] The number of Query attention heads that share one KV head [5].

[2] Activations occupy negligible (less than 5%) of the memory [17], thus we omit them for simplicity.

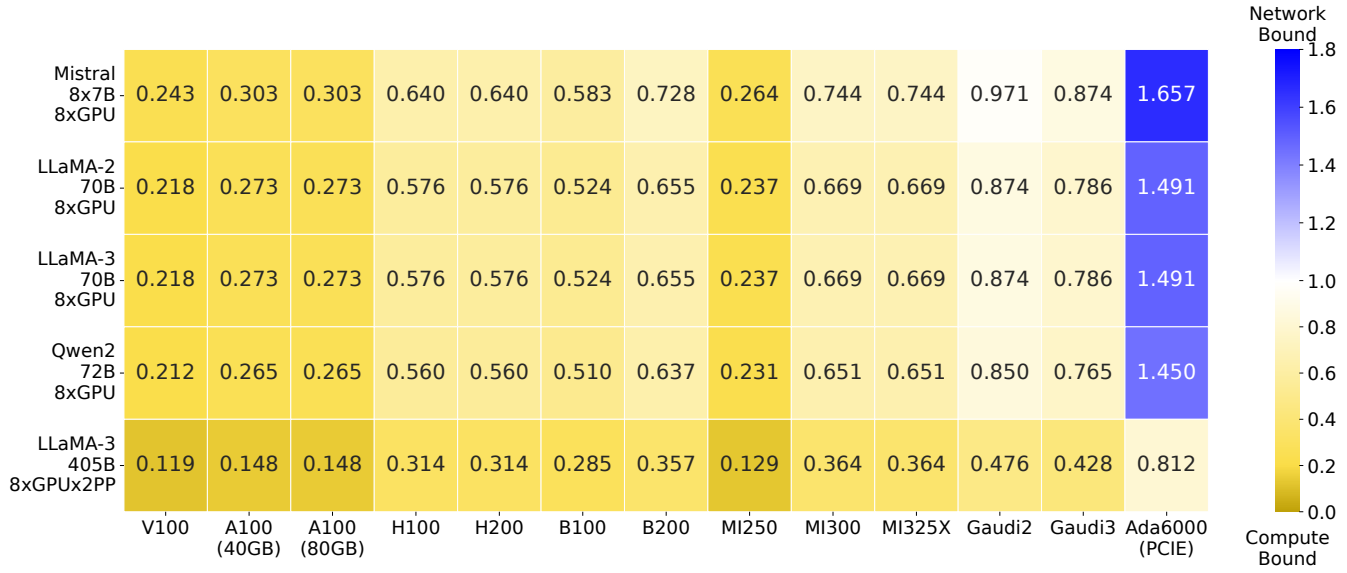


Figure 2: Comparison of network time and compute time. The closer to yellow, the more compute-bound the workload is, whereas the closer to blue indicates the workload is more network-bound.

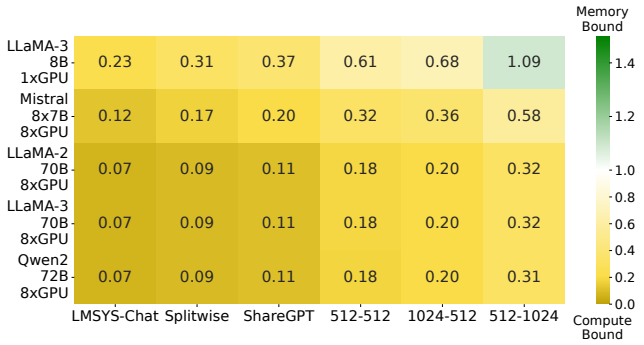


Figure 3: Comparison of compute time and memory time. The closer to yellow, the more compute-bound the workload is, whereas the closer to green the more memory-bound it becomes.

dense operations is $2B_{Dense}\sum N_w K_w$. Note that the $\sum N_w K_w$ term is the number of weight elements in all layers, which can be approximated using P_{Model} (the parameters in the model), thus, we simplify the expression as $2B_{Dense}P_{Model}$. Therefore, latency from the compute perspective is:

$$T_{Compute} \approx \frac{2B_{Dense} \cdot P_{Model}}{Compute} \tag{2}$$

Network. Finally, for tensor parallelism, collective network communication primitives are needed to synchronize the results after performing operations in multiple GPUs over different shards of data. The size of the data transmitted with collective communication equals the size of inputs for dense operations, which have dimensions $[B_{dense}, D_{model}]$. To support tensor parallelism, two AGs and one AR (or 2 ARs) are required [39]. An AR requires gathering outputs from other GPUs, performing local summation, and broadcasting results, thus it needs to transfer the activations twice, while an AG transfers activations once. Therefore, in both cases, we can estimate the total data movement for one GPU in bytes as $4 \cdot B_{Dense} D_{model} S_{type} \cdot L$. Thus, the latency from the network perspective is:

$$T_{net} \approx 4 \cdot \frac{(N_{GPU}-1) B_{Dense} D_{model} S_{type} L}{NetBW} \tag{3}$$

3.3 Classification of LLM Serving Workloads

We classify LLM serving workloads by comparing the latency of an iteration derived based on memory, compute, and network requirements.

First, we compare Equation 3 (network time) with Equation 2 (compute time). Their ratio can be calculated as

$$\frac{T_{Net}}{T_{Compute}} = \frac{2D_{model}L}{P_{model}} \frac{(N_{GPU}-1)Compute}{NetBW/S_{type}}$$

Note that $P_{model} \approx 12D_{model}^2 L$[3], thus $\frac{2D_{model}L}{P_{model}} \approx 1/(6D_{model})$. For models that need multiple GPUs (where T_{Net} is relevant), D_{model} is typically larger than 4096 [9, 16, 22, 48], making the term $\frac{2D_{model}L}{P_{model}}$ smaller than 1×10^{-5}. Due to the high bandwidth interconnect between data center GPUs (e.g., NVLink [25], Infinity Fabric [54]),

[3] $P_{model} = ((W_K + W_Q + W_V) + W_O + W_U + W_G + W_D)L \approx ((1/R_{GQA} + 1 + 1/R_{GQA})D_{model}^2 + D_{model}^2 + D_{model}I_{model} + D_{model}I_{model} + I_{model}D_{model})L) \approx (1+1+3.5+3.5+3.5)D_{model}^2 L \approx 12D_{model}^2 L$, where I_{model} is intermediate dimension, which is typically 3.5x of D_{model} [16, 22].

Table 1: Characteristics of accelerator models across various vendors and release years.

Vendor	Model	Release Year	MemSize (GB)	MemBW (GB/s)	NetBW (GB/s)	Compute (FP16 GFLOP/s)	$\frac{MemSize}{MemBW}$	$\frac{Compute}{MemBW}$	$\frac{NetBW}{MemBW}$
NVIDIA	V100	2017	16	900	300	125,000	0.018	139	0.33
NVIDIA	A100	2020	40	1,555	600	312,000	0.026	200	0.39
NVIDIA	A100	2021	80	2,000	600	312,000	0.040	156	0.30
NVIDIA	H100	2023	80	3,352	900	989,000	0.024	295	0.268
NVIDIA	H200	2024	141	4,800	900	989,000	0.020	206	0.19
NVIDIA	B100	2024	192	8,000	1,800	1,800,000	0.015	225	0.23
NVIDIA	B200	2024	192	8,000	1,800	2,250,000	0.015	281	0.23
AMD	MI250	2021	128	3,352	800	362,000	0.038	107	0.24
AMD	MI300	2023	192	5,300	1,024	1,307,000	0.036	246	0.19
AMD	MI325X	2024	256	6,000	1,024	1,307,000	0.043	218	0.17
Intel	Gaudi 2	2022	96	2,400	600	1,000,000	0.040	417	0.25
Intel	Gaudi 3	2024	128	3,700	1,200	1,800,000	0.035	486	0.32
NVIDIA	Ada 6000	2022	48	960	64	182,000	0.050	190	0.067

$\frac{N_{GPU}Compute}{NetBW/S_{type}}$ ranges from 1×10^4 to 1×10^5. Therefore, this ratio is typically less than 1, indicating that the network is not the bottleneck. We illustrate this ratio in Figure 2. For large models and data center GPUs, the compute takes more time than network.

We then compare Equation 1 (memory time) and Equation 2 (compute time) to finally determine the characteristics of the workload, namely, the following value

$$T_R = \frac{T_{Mem}}{T_{Compute}} \approx \frac{Compute}{MemBW}\frac{MemSize}{P_{model}}\frac{1}{2B_{dense}} \tag{4}$$

Modern models widely adopt Grouped Query Attention (GQA) as seen in LLaMA-3 [48], NeMo [4], and Qwen2 [9]. In GQA, multiple attention heads share a KV-cache, which effectively means that given the same memory capacity, batches can contain more requests. Consequently, B_{dense} tends to be larger. For example, when serving LLaMA-2 70B on $8\times A100$ GPUs, the maximum batch size of decode requests is on the order of 1024. Combined with the prefill tokens in the batch, B_{dense} can reach 2048 while a non-GQA model with the same size can only have $B_{dense}=256$. Thus, the ratio T_R is less than 1, and compute becomes the main bottleneck. Additionally, modern models tend to have increasingly large model sizes P_{model} (whereas the rest of the terms in T_R are constants given a hardware specification), further causing T_R to drop below 1.

Figure 3 shows the values of T_R with five representative models: LLaMA-3 8B [22] with 1×A100, Mistral 8×7B [16] with 8×A100, LLaMA-2 70B [49] with 8×A100s, LLaMA-3 70B with 8×A100s, Qwen-2 72B [9] with 8×A100s. As the heatmap figure shows, many of the workloads (Splitwise [32], LMSYS-Chat-1M [56], ShareGPT [1] as well as constant length inputs and outputs), are uniformly compute-bound, except in the case of the long decode (512-1024) workload on LLaMA-3 8B model, where the ratio T_R is around 1.

While the analysis is mainly conducted on an NVIDIA A100, the result remains similar for other accelerators. As shown in Table 1, compute/memory ratio and compute/network ratio are stable across vendors and generations, meaning the workload characteristics also remain largely unchanged.

To summarize, our cost model suggests that modern LLM serving workloads operate in a compute-bound regime, which we validate empirically in the next section.

3.4 Validation of the Cost Model

We validated our cost model on LLaMA-2 70B serving with 8 NVIDIA A100 GPUs and a dense batch size of 2048 requests. We computed the GFLOPs, memory movement, and network traffic for each operation (Table 2) and used these with our model to calculate $T_{compute}$, T_{mem}, and T_{net} based on hardware specifications[4]. The longest estimated time $T_{op}=\max(T_{compute},T_{mem},T_{net})$ indicates the most constrained resource. To validate these results, we measure the running time of different operations. For most operations, T_{op} aligns with profiling results. An exception is prefill attention, where the real measured time is larger due to the launching overhead of the associated kernels dominating the total time. The sums of all operations' $T_{compute}$, T_{mem}, and T_{net} values show that compute is the most constrained resource, which aligns with our model's finding.

3.5 Optimal Serving Throughput

Optimal throughput (measured in token/s/GPU) is achieved when the most limited resource, i.e. compute, is fully utilized. By applying Equation 2, the system's optimal throughput is:

$$\text{Throughput}_{\text{optimal}} = \frac{B_{Dense}}{T_{Compute}} = \frac{Compute}{2P_{Model}} \tag{5}$$

In the compute-bound regime, the optimal throughput depends *solely* on the aggregate computational capacity of the GPUs for the specific data type and the number of parameters in the model. Notably, other factors, including the GPU memory size, bandwidth, model data type, or the length of prefill and decode phases, do not have a significant impact on

[4]One-way network bandwidth was used for T_{net}

Table 2: Comparison of operation runtimes between cost model estimation and real-world measurements.

Operation	Compute (GFLOP)	Mem Load (GB)	Net Usage (GB)	Est. T_{comp} (ms)	Est. T_{mem} (ms)	Est. T_{net} (ms)	Real Time (ms)
KQV	27487.8	19.5	0	**11.01**	1.22	0	**16.08**
O	21990.2	16.1	0	**8.81**	1.01	0	**16.01**
UG	153931.6	96.6	0	**61.67**	6.04	0	**69.92**
D	76965.8	49.7	0	**30.84**	3.11	0	**34.96**
DecAttn	3665.9	462.2	0	1.47	**28.89**	0	**35.60**
PfAttn	916.3	2.1	0	**0.37**	0.13	0	**4.56**
Net	18.8	75.2	75.2	0.01	4.70	**31.33**	**47.92**
Total				**114.17**	45.09	31.33	

the theoretical optimal throughput, as long as the workload is overall compute-bound. We measure the GPU compute capacity by profiling the state-of-the-art GEMM vendor library, CUTLASS [46]. For example, when serving LLaMA-2 70B on 8×A100 GPUs, the profiled peak *Compute* is 280 TFLOPS for FP16, and the P_{Model} is 70B. Substituting these into Equation 5, yields an optimal throughput of 1857 tokens/s/GPU.

3.6 Gap to Optimal Throughput

As shown in Figure 7, existing serving systems fall far short of achieving optimal throughput. When serving offline requests, vLLM, DeepSpeed-FastGen, and TensorRT-LLM achieve only 22.0%, 22.9%, and 37.8% of the optimal throughput, respectively.

The main reason why existing serving systems fall significantly short of achieving optimal throughput is poor utilization of GPU resources. Figure 4 shows the execution flow of transformer operations within a GPU, using existing serving frameworks like SGLang [58], vLLM [17], DeepSpeed-FastGen [13]. These frameworks execute compute-, memory-, and network-bound operations sequentially on a single large batch in a device. This sequential execution flow forms multiple pipeline bubbles (denoted as "WASTED" in Figure 4), which cause the most constrained resource, compute, to be underutilized.

3.7 Intra-device Parallelism

Ostensibly, the advantage of existing serving frameworks when using a single large batch in a GPU is that they need to load model weights fewer times than if there were multiple smaller batches and corresponding operations.

However, the cost model and its validation that we outlined earlier in this section provides a valuable insight: Since modern models are largely compute-bound (by a factor of more than 2 as shown in Figure 3 and Table 2), creating and processing smaller batches may be justified, especially if compute utilization can be increased.

These observations motivate intra-device parallelism through nano-batching, which underlies the design of NanoFlow. NanoFlow creates multiple nano-operations, which are the same operations as the original operations, but operate on finer-grained batches that we call nano-batches. For example, for the original Up projection operating on a batch size of 2048, NanoFlow could create two nano-operations UP1 and UP2, operating on batches in ranges 0-768 and 768-2048, respectively, while performing the exact same projection. Different nano-batches do not have data dependencies. Therefore, nano-operations that use different resources—compute, memory, network—can operate on different nano-batches in parallel within a device to maximize the utilization of the most constrained resource, compute. Despite requiring repeated loading of model weights, we demonstrate that NanoFlow provides significantly more throughput than existing frameworks (1.91× on average), further closing the gap to optimal serving throughput (§6).

4 Design

We introduce NanoFlow's key components: the auto-search engine for automatic nano-batch pipeline computation (§4.1) and the LLM serving runtime for pipeline execution (§4.2).

4.1 Automated Pipeline Search

Deciding number of nano-operations, kernels, scheduling with intra-device parallelism, and resource allocation is challenging due to model and hardware diversity. To address this, NanoFlow proposes auto-search, which uses Mixed Integer Linear Programming (MILP) to obtain the nano-batch overlapping strategy, with a two-stage approximation for reducing the search space. We first outline auto-search prerequisites including kernel profiling and interference modeling (§4.1.1), then detail its two stages: pipeline structure search (§4.1.2) and GPU resource allocation (§4.1.3). Finally, we provide example pipelines for popular models (§4.1.4).

4.1.1 Kernel Profiling and Interference Modeling

Understanding the performance characteristics of different operations is key to automated pipeline search. For this, NanoFlow profiles GEMM kernels (used for compute-bound operations such as KQV generation), GEMV kernels (used in memory-bound operations as in decode-attention), and network kernels (used in network-bound operations such as AR and AG).

Determining the maximum dense batch size Given a specific model architecture and user input statistics (as described in §3.1), NanoFlow calculates the maximum *dense batch size* that can fit into the GPU memory (e.g., around 2048 for $8 \times A100$ serving LLaMA-2 70B). This dense batch size serves as an input to auto-search, which sets an upper bound of shapes in profiling.

Profiling interference-free kernels NanoFlow then determines the interference-free execution time of kernels, where each operation exclusively uses the GPU. NanoFlow profiles kernels with discrete input batch sizes from 128 to the dense batch size in multiples of 128, as 128 is a hardware-friendly shape for GEMM tiling. NanoFlow explores all possible kernel implementations varying the number of thread blocks, the

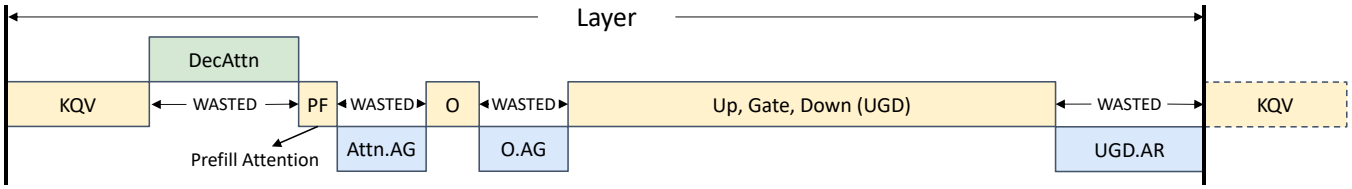


Figure 4: Execution pipeline of existing systems. The green, yellow, and blue operations correspond to memory-, compute-, and network-bound operations. Operations in the previous and next layer are denoted by dotted borders. "WASTED" shows the stages in the pipeline where the most constrained resource, compute, is underutilized. Small operations (i.e. layernorm, activation, etc.) are omitted for simplicity.

number of warps, and tile size for GEMM, GEMV, and network kernels to identify the implementation that provides the shortest execution time. The profiling outputs a mapping from (kernel, batch size) to its best implementation and execution time.

Profiling kernels with interference However, when two or more kernels execute in parallel on a device, they slow down compared to running separately. We refer to this as *kernel interference*, which is attributed to kernels competing for GPU hardware resources such as execution units, caches, power, and memory controllers [43]. Ideally, we would like to independently assign each nano-operation a fractional allocation of a single GPU's compute, memory, and network bandwidth. We call this fraction $R_{physical}$. Unfortunately, kernel interference is unpredictable, as NVIDIA GPUs do not give explicit control over compute, memory, and network bandwidth. Thus, we cannot directly control $R_{physical}$.

Instead, we use GEMM performance as a proxy R for $R_{physical}$ and model pairwise kernel interference. We define R in a GEMM-centric way since the compute-bound operations dominate and optimizing compute is the goal of NanoFlow.

In particular, we profile pairwise kernel interference patterns by measuring actual performance when overlapping between a compute kernel A and another kernel B. For example, if when overlapped, GEMM kernel A achieves 40% performance compared to its best performance when run individually, then we say that R_A=0.4. We then assume that the overlapped kernel is given the remaining resources, so we set $R_B = 1 - R_A$ =0.6.

Then, we model the interference by measuring the performance loss in the non-compute kernel B when R_A resources are allocated to kernel A. When overlapping A and B, we normalize A and B's performance to their peak performance when run individually. We call this value P. For the compute kernel A, P_A is equivalent to R_A by definition, while P_B captures the memory or network utilization when R_A resources are assigned to the compute kernel.

Intuitively, this process establishes an exchange rate between compute utilization R and memory or network performance P . The overall kernel interference model can then be captured in a table such as Table 3, which maps R to P for each kernel type.

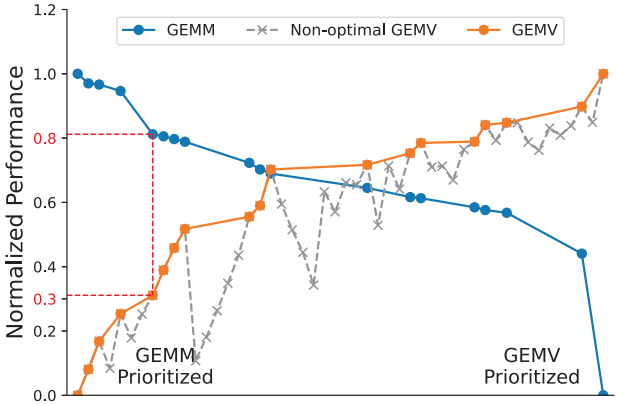


Figure 5: Interference characteristics between GEMM and GEMV kernels. The points on the x-axis correspond unique GEMM-GEMV implementation pairs. The y-axis denotes the GEMM and GEMV kernels' normalized performance P.

Nonetheless, profiling the mapping between R and P remains challenging due to the vast number of possible kernel combinations. While each operation may have only a few hundred kernel implementations, the combinations between different implementations of overlapping kernels exponentially expand the profiling space, resulting in millions of possible configurations. This immense complexity makes exhaustive exploration infeasible.

NanoFlow reduces the profiling space by limiting the thread block numbers for GEMV and network kernels to values between 8 and 128 (in steps of 8), as 128 blocks are sufficient to saturate their performance. It also excludes less efficient GEMM kernels, which have longer execution times while using more thread blocks. NanoFlow focuses on pairwise interference analysis—compute-memory and compute-network—rather than examining three kernel types simultaneously and assumes that the R to P mapping profiled with pairwise interference holds when overlapping three kernels.

Figure 5 shows NanoFlow's profiling results of GEMM-GEMV kernel pairs ($\sim$ 100 pairs, after the aforementioned simplifications) on an A100 GPU with GEMM shape $(M,N,K) = (384,4096,4096)$, GEMV batch size 384, and

Table 3: Performance P of GEMV and network kernels with different resource utilization R.

Operations	Resource Utilization (R)						
	0	0.1	0.2	...	0.8	0.9	1
GEMM (by definition)	0	0.1	0.2	...	0.8	0.9	1
GEMV	0	0.2	0.3	...	0.85	0.95	1
Network	0	0.3	0.5	...	0.9	1	1

sequence length 1024. These pairs are then sorted by descending GEMM performance, as shown in Figure 5. For instance, a normalized performance of 0.8 means the kernel's execution time is $1/0.8$ times longer than the fastest implementation. Pairs with higher interference on GEMM but worse GEMV performance (grayed-out in Figure 5) are discarded. This process identifies the best-performing kernel combinations at various GEMM-GEMV performance trade-off points.

Using interference profiles, auto-search generates a *resource mapping table* (Table 3). This table quantifies GEMV and network kernel performance (P) as functions of resource utilization (R). For example, as depicted by the dotted red line in Figure 5, achieving 0.3 GEMV performance requires sacrificing 0.2 GEMM performance (from 1 to 0.8). This implies a non-linear trade-off: 0.2 GEMM performance is exchanged for 0.3 GEMV performance, indicating that $R_{\text{GEMV}} = 0.2$ corresponds to $P_{\text{GEMV}} = 0.3$.

A sensitivity analysis of all GEMM shapes in the evaluated LLM models and 64 batch size combinations reveals that the R to P mapping is consistent, with a standard deviation within 5% of the mean. Consequently, Table 3 can be used to map R to P for all GEMV and network kernel implementations in subsequent auto-search analysis.

4.1.2 Auto-search Stage I: Pipeline Structure Search

In the first stage, auto-search takes as input the dense batch size, the operation dependencies (defined in the PyTorch implementation of a given model architecture), and the profiling of interference-free kernels, and then produces as output, the number, batch size, and order of each nano-operation using mixed integer linear programming (MILP). Stage I does not factor in the interference between nano-operations to simplify the problem, which is covered by Stage II. The MILP problem is subject to the constraints described below.

Optimization objective. The main objective of MILP is to remove pipeline bubbles (denoted by wasted in Figure 4) for compute operations to minimize execution time.

Constraints on the number of nano-operations. To overlap resources, each operation needs to be split into at least two nano-operations, each operating on an independent nano-batch. As splitting into nano-operations reduces the batching effect, the number of nano-operations should be minimized. Thus, auto-search begins its search by dividing all operations into two nano-operations and formulating the MILP problem to identify the batch size, execution time, and ordering of the nano-operations. If the temporary best solution has bubbles for compute, auto-search increases the number of nano-operations for operations near the bubble to improve resource utilization until MILP cannot produce better solutions.

Constraints on batch sizes and execution times. Auto-search forces the batch sizes to be chosen from 128 to dense batch size and uses the interference-free execution time of a nano-operation for a given batch size, as computed in §4.1.1.

Constraints on dependencies. The dependencies of nano-operations are determined by their parent operations (present in baseline implementations, e.g., in pytorch) and their input batches. Two nano-operations are dependent if and only if their parent operations are dependent (e.g., an O-projection depends on an attention operation) and their input batches intersect (e.g., ranges 0–255 and 128–383).

Constraints on overlapping. Overlapping kernels constrained by the same resource (e.g., compute) are unhelpful for utilization, so auto-search restricts the pipeline to overlap only operations constrained by different resources.

Constraints on operation transformations. Some network collectives have equivalent transformations with different performance characteristics and dependencies. For example, an AG can be converted into an AR with different weight partitioning approaches [39, 57]. NanoFlow explores all of these alternative transformations for network nano-operations and searches for the shortest pipeline design.

Due to the large search space, searching for an optimal pipeline would take hours or even days. However, a practical pipeline can be found in approximately 10 minutes by prioritizing searching a feasible solution instead of finding a provably-optimal solution.

4.1.3 Auto-search Stage II: Refining the Pipeline

The pipeline derived in the first stage does not factor in kernel interference, which is impractical for real deployment. In the second stage, auto-search refines this pipeline by considering kernel slowdown due to interference. This stage formulates a separate MILP problem, where the numbers, batch sizes, and ordering constraints of nano-operations remain the same (i.e., outputs of the first stage). Auto-search explores various resource utilization (R) allocations to individual nano-operations, and maps R to P using Table 3 derived in interference profiling. A specific resource utilization value R directly corresponds to kernel implementations as determined in §4.1.1. The goal of the second stage is again to minimize pipeline execution time.

Constraints on GPU resource utilization. As discussed in §4.1.1, concurrent kernels should compete for a total of 1.0 of GPU resource utilization R. Thus, auto-search forces the sum of R at any given time to be less than or equal to 1.0.

Constraints on execution times. Given the resource utilization R for a nano-operation, auto-search computes the execution time of the operation using D_{best}/P, where D_{best}

represents the best execution time without interference and P is derived using Table 3.

The auto-search is performed only when the model architecture or workload (input length, output length) undergoes significant changes. Compared to the long-running deployment times, the search time is negligible.

4.1.4 Example Pipelines

70B pipeline We apply auto-search to 70B-scale models, including LLaMA-2 70B, LLaMA-3 70B, Qwen2.5-72B, and Deepseek-67B. Compared to LLaMA-2 70B, LLaMA-3 70B features a larger vocabulary size of 128K, which increases the sampling time; Deepseek-67B has a slightly different number of layers and hidden dimension (D_{model}), Qwen2.5-72B introduces bias in KQV generation. However, none of these adjustments significantly change the performance characteristics, leading to similar schedules. We provide the NanoFlow-generated pipeline for LLaMA-2 70B as an example in Figure 6. Since three resources (compute, memory, and network) overlap at the beginning of a decoding layer (i.e., KQV generations), auto-search ends up using 4 nano-operations for this part, to reduce pipeline bubbles. Here, the decode attention operation has resource utilization 0.4 (i.e. reduces 40% GEMM performance) and reaches 80% of maximum decode attention performance. For the remainder of the pipeline, GEMM operations are prioritized, and only two nano-operations are used.

8B pipeline 8B models (e.g. Llama3-8B) do not require network operations as they fit in a single GPU. Thus auto-search splits all operations into two nano-operations, with decode attention overlapping with the Up Gate Down projection.

MoE pipeline Compared to 70B models, MoE models have different hidden dimensions and number of layers. Due to the imbalance between experts [16], NanoFlow uses tensor parallelism for MoEs, in which FFN is implemented using grouped-GEMM. FFN also involves an additional gate routing operation. Auto-search works similarly for an MoE model and automatically produces an efficient pipeline (§6).

4.2 NanoFlow Runtime

We now explain how NanoFlow executes the auto-generated pipeline by asynchronous scheduling (§4.2.1), and KV-cache SSD offloading for multi-round conversations (§4.2.2).

4.2.1 Request Scheduling

Batch formation NanoFlow assumes that auto-scaling, workload balancing, and priority-aware routing are managed by the control plane externally [15,33,45]. A NanoFlow instance assumes an abundance of requests and treats each request with equal priority. When requests are not abundant, the control plane should reduce the number of NanoFlow instances to maintain a sufficiently large per-instance batch size. Note that while the B_{dense}, i.e. the token batch size for GEMMs, is on the order of thousands (e.g., 1 prefill request of 2048 tokens and 256 decode tokens), this represents the combination of prefill and decode tokens. Therefore, the request batch size (e.g., 1 prefill and 256 decode requests) is on the order of hundreds, which is easy to attain in real-world large-scale serving.

When forming a batch, NanoFlow prioritizes unfinished decode requests and chunks prefill requests at a token granularity following SarathiServe [3] to exactly fill the remaining capacity of a pre-selected best-performing dense batch. This effectively reduces the tail latency as dense operations are performed over consistent batch sizes across iterations. Initially, the global batch contains only prefill requests (requests in prefill stages). As these requests complete the prefill stage, they transition to decode requests within the batch, and new prefill requests are introduced to maintain the fixed token batch size. As decode requests increase, the prefill token budget decreases, causing prefill requests to slow down. When decode requests begin completing, this frees space for more prefill tokens again, speeding up prefill. Thus, the number of decode requests and prefill requests will be stabilized.

To optimize GPU memory usage and avoid running out of memory, NanoFlow predicts peak future memory usage based on the status of user requests. NanoFlow tracks decoded tokens per request, estimates completion time using average decode length, and prefills new requests only if predicted memory usage stays within GPU limits. If the GPU runs out of memory, NanoFlow move a request to the CPU and reloads it once memory is available without recomputation.

Asynchronous scheduling Batch formation, including estimating memory usage, scheduling new requests, retiring finished requests, and adjusting the page table for PagedAttention [17], consumes a non-negligible amount of time on the CPU side [42]. In most serving frameworks [17,58], *only after* the GPU executes one iteration, the scheduler on the CPU is able to detect *EOS* tokens, remove the finished request from the batch, and refill the batch with new requests. However, GPU is under-utilized during this time. To avoid this waste, NanoFlow asynchronously schedules batch formation in parallel to the GPU executions. At any iteration i, NanoFlow forms the batch for the next iteration *before* the end of the current iteration. This means that NanoFlow cannot detect the *EOS* tokens generated at iteration i. After launching iteration $i+1$, NanoFlow forms the batch for cycle $i+2$, detects the *EOS* token from iteration i, and removes finished requests. Fortunately, since the average decode length surpasses 100 for typical workloads (See Table 4), the overhead of one extra decode token is negligible ($< 1\%$), given the benefit of hiding the batch formation overhead.

4.2.2 KV-cache Management

To efficiently serve multi-round applications, NanoFlow offloads the KV-cache of running requests to a hierarchical cache consisting of host CPU memory and SSDs, and loads a prior round's KV-cache when a new round arrives.

Simultaneous offloading. Instead of waiting for requests'

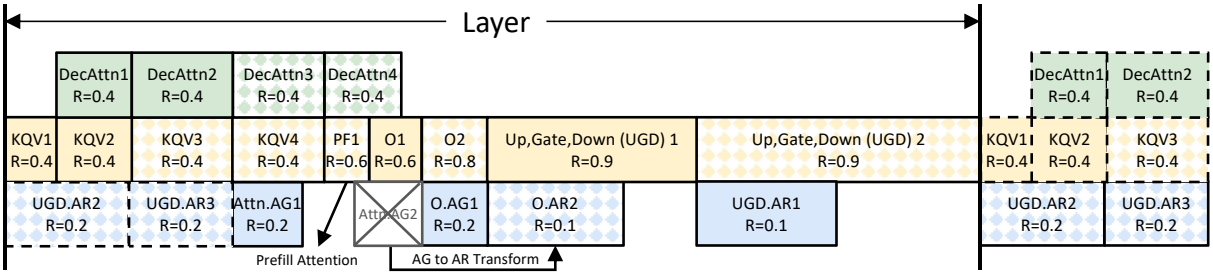


Figure 6: Execution pipeline of LLaMA-2 70B, automatically generated by NanoFlow. The solid background and shaded background represents input batch 0-768 and 768-2048, respectively. *R* stands for resource utilization. By overlapping the compute-, memory-, and network-intensive operations, NanoFlow increases compute utilization and improves the serving throughput.

completion, NanoFlow offloads the KV-cache of tokens directly after the KQV generation in each transformer layer before appending them to the KV-cache. The KV vectors output by KQV generation are naturally contiguous and the data size of the offload is balanced across iterations by design as NanoFlow's dense batches remain stable. As a result, the host and the GPU hold the same copy of on-the-fly requests' KV-cache. To minimize the offloading overhead, NanoFlow performs the device-host copy— an operation that uses minor GPU resources—during compute-bound operations in FFN. Moreover, NanoFlow reduces offloading time via NUMA-aware thread-binding.

Host KV-cache management. NanoFlow uses the LRU policy to manage the hierarchical cache of CPU memory and SSDs. For e.g., KV-cache are evicted to SSDs when the CPU reaches its memory limit and retrieved from either CPU or SSD to initiate the loading process when the next round conversation of a request comes in.

KV-cache loading and scattering. NanoFlow uses PagedAttention [17], therefore, the pages of the KV-cache reside in the GPU memory in a fragmented manner. In order to avoid copying to fragmented GPU page destinations, NanoFlow first copies the data to a contiguous space on GPU and then scatters the pages to their destinations. This achieves 7-10× higher bandwidth for host-to-device copy.

5 Implementation

We implemented NanoFlow for NVIDIA GPUs. NanoFlow consists of ∼10K lines of CUDA and ∼6K lines of Python.

Given the GPU resource requirements determined by auto-search, NanoFlow knows which kernels to launch for each nano-batch to achieve a given utilization level, based on its profiling results. NanoFlow launches nano-operations, respecting the ordering constraints from auto-search, on multiple CUDA streams and enforces ordering dependencies using CUDA events.

6 Evaluation

We now mainly answer the following questions:

Dataset	Avg. Input (Std)	Avg. Output (Std)
Splitwise [32]	1155 (1109)	211 (163)
LMSYS-Chat [56]	102 (169)	222 (210)
ShareGPT [1]	246 (547)	322 (244)

Table 4: The average and standard deviation of input and output lengths in the sampled datasets.

- How much NanoFlow improves serving throughput compared to existing systems, and how does its throughput compare to the optimal serving throughput? (§6.2)
- What is the average serving latency under NanoFlow for different request rates, and how is it distributed? (§6.3)
- How do the various techniques proposed in NanoFlow contribute to the end-to-end throughput? (§6.4)
- What is the compute, memory and network resource usage pattern of NanoFlow? (§6.5)
- How does NanoFlow improve performance when applied to different popular LLMs? (§6.6)

6.1 Experiment Setup

Hardware. We run our experiments on 8× A100 80GB SXM GPUs interconnected via NVLink.

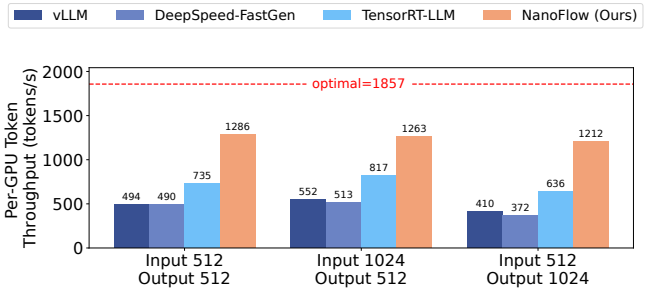
Models. We provide a detailed evaluation of NanoFlow using the LLaMA-2-70B model, one of the most widely-used open-source LLMs. We also demonstrate the applicability of NanoFlow by evaluating it on 5 other representative LLMs. We use FP16 weights and activations for all models, as this is standard for data center-scale inference.

Baselines. We consider three widely-used serving frameworks as baselines.

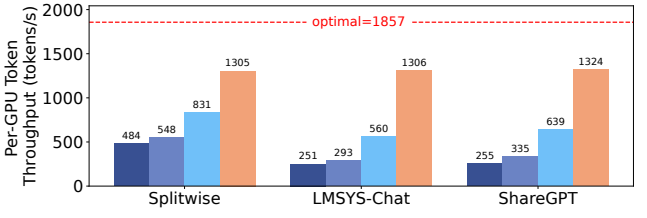
vLLM[5] [17, 52] is a state-of-the-art serving system delivering high throughput. vLLM implements pagedAttention for increasing GPU memory utilization, as well as the chunked prefill for higher GPU compute utilization.

DeepSpeed-FastGen[6] [13, 23] is a serving framework de-

[5] v0.5.3.post1 (commit ID 38c4b7e)
[6] v0.2.3 (commit ID 429bc5c)



(a) LLaMA-2-70B, 8 GPU, TP=8, Constant Input & Output Length



(b) LLaMA-2-70B, 8 GPU, TP=8, Input & Output Length from Dataset

Figure 7: Offline throughput comparison. NanoFlow outperforms all baselines for all the workload settings. TP stands for the number of GPUs used with tensor parallelism.

veloped by Microsoft. It dynamically composes prefill with decode requests to ensure that the engine is operating in a high throughput regime. We vary the `max-ragged-batch-size` to tune the batch size for highest throughput.

TensorRT-LLM[7] [26, 27] is a high-performance LLM inference engine built upon NVIDIA's TensorRT SDK. We set `max-num-tokens` by calculating the maximum capacity for the KV-cache in the GPU memory. We also enable paged KV-cache and dynamic batching optimizations when compiling.

Datasets. Splitwise [32] is a conversation trace collected from a real production environment at Microsoft, with a total of around 20000 requests. LMSYS-Chat-1M [56] is a large-scale dataset with 1 million real-world conversations from 25 different LLMs. ShareGPT [1] is a dataset with conversations collected from the ShareGPT API. We use the full trace from Splitwise and randomly sample 50,000 requests from ShareGPT and LMSYS-Chat-1M for our evaluation. Table 4 shows the average input length and output length in tokens for the sampled datasets we use.

6.2 Throughput

We first evaluate throughput to simulate use cases such as benchmarking, information extraction, data wrangling, and form processing [36]. We sample actual input (prompt tokens) and output (generated tokens) lengths from the datasets and add additional experiments with constant length. We then deploy a NanoFlow instance (with a dense batch size of 2048 in the case of Llama-2-70B, where NanoFlow delivers best performance) to measure the *total throughput* defined as the total number of input and output tokens divided by the execution time. We compare the performance of the NanoFlow instance with optimal throughput derived using Equation 5, where we measure the peak compute capacity using the state-of-the-art GEMM vendor library CUTLASS with a token batch size of 2048. Since the optimal throughput is independent of user query statistics, for all the experiments, the optimal throughput is 1857 tokens/s/GPU as computed in §3.5.

Figure 7 shows the throughput of the NanoFlow instance compared to the baselines. NanoFlow has the highest throughput in all settings and is able to achieve 68.5% of the theoretical optimal throughput in the best case. For constant lengths, NanoFlow has on average 2.62× higher offline throughput than vLLM, 2.78× higher than DeepSpeed-FastGen and 1.73× higher than TensorRT-LLM. When input and output lengths are drawn from the datasets, NanoFlow has on average 4.18× higher throughput than vLLM, 3.45× higher than DeepSpeed-FastGen and 1.91× higher than TensorRT-LLM.

6.3 Latency

We evaluate the latency of the NanoFlow instance, for which we model the request arrival interval via an exponential distribution, following prior work [17]. We generate 5 minutes of request traces from the datasets in Table 4 for various request rates. We then measure *normalized latency* by first calculating the end-to-end request latency divided by output length in tokens, then taking the average for all requests. For each framework, we gradually increase the request rate and measure the normalized latency. We select the SLO for average normalized latency as 200*ms* following prior works [59], which is also the typical human reading speed [8].

Figure 8 demonstrates the normalized latency of the NanoFlow instance compared to baselines at various request rates. At lower rates, the NanoFlow instance has comparable but slightly higher latency, because NanoFlow targets throughput-oriented scenarios and therefore employs a large dense batch size.

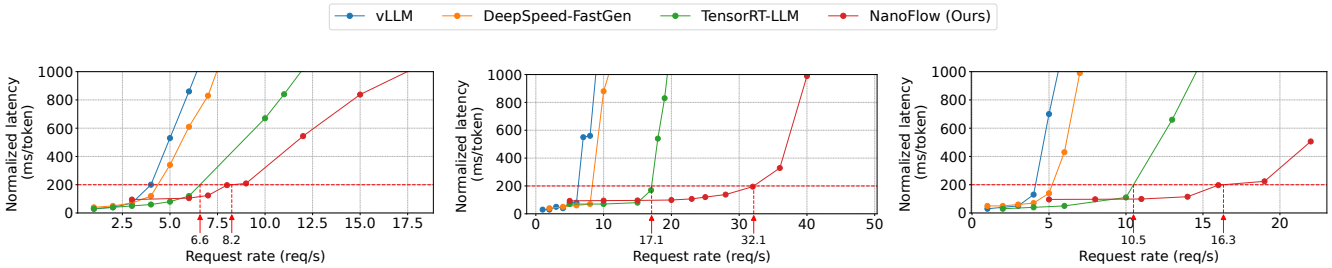
The NanoFlow instance is able to sustain a higher request rate within 200*ms* latency SLO compared to all other baselines across all the datasets. For example, for LMSys-Chat-1M dataset, the NanoFlow instance can achieve 1.64× higher request rate compared to TensorRT-LLM within 200ms normalized latency constraint.

Moreover, the NanoFlow instance's 99th-percentile latency is only 1.07× of the average latency at near-maximum throughput, as NanoFlow uses a constant dense batch size, helping it perform well even under tail latency SLOs.

6.4 Ablation Study

To showcase the overhead and benefit of the techniques that NanoFlow uses, including nano-batching, nano-operation overlapping, and KV-cache offloading, we compare the NanoFlow instance with baselines that share the same asyn-

[7] v0.8.0 (commit ID 5955b8a)



(a) Splitwise (b) LMSYS-Chat-1M (c) ShareGPT

Figure 8: Latency comparison. The x-axis shows the number of incoming requests per second, and the y-axis shows the normalized latency. NanoFlow handles higher request rates within 200ms SLO constraints.

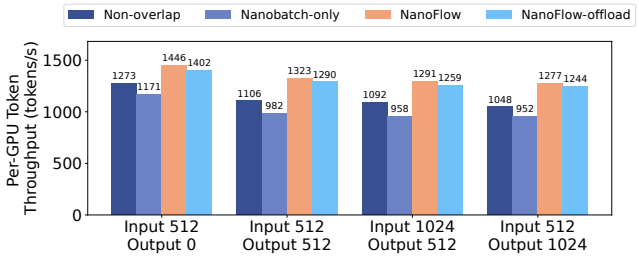


Figure 9: Ablation study results for NanoFlow. Nano-batching and overlapping improves NanoFlow's performance.

chronous request scheduling and kernel libraries.

We created two baselines: a non-overlapping baseline processing inputs sequentially without nano-batches, and a nano-batch-only baseline that splits requests into nano-batches as in NanoFlow but executes them sequentially to assess nano-batching overhead. As shown in Figure 9, splitting into nano-batches alone reduces performance by 13.2%.

To estimate the benefit of overlapping network- and memory-bound kernels, we compare NanoFlow and baselines under various prefill-decode ratios, as shown in Figure 9. We use prefill-only workloads (Input 512, Output 0) to demonstrate the benefit of overlapping network-bound and compute-bound kernels. Additionally, we use decode heavy workloads (Input 512, Output 1024) to evaluate the benefit of overlapping both network- and memory-bound kernels. NanoFlow achieves 1.07× speedup by overlapping network-bound kernels and 1.17× speedup by overlapping both network- and memory-bound kernels compared with the non-overlapped baseline.

Moreover, we quantify the performance degradation due to offloading in Figure 9. Enabling offloading would slow down the pipeline by 3.0% due to kernel interference caused by KV-cache movement. However, offloading can reduce compute by 3.02× for multi-round LMSYS-Chat workloads.

6.5 Resource Usage

We demonstrate NanoFlow's effectiveness in terms of resource utilization in Figure 10. While the non-overlapping baseline sequentially executes operations, which mostly uses only one resource at a given time, the NanoFlow instance can concurrently utilize multiple resources and achieves 68.5% average compute utilization. Due to kernel interference, NanoFlow provides lower than optimal compute usage.

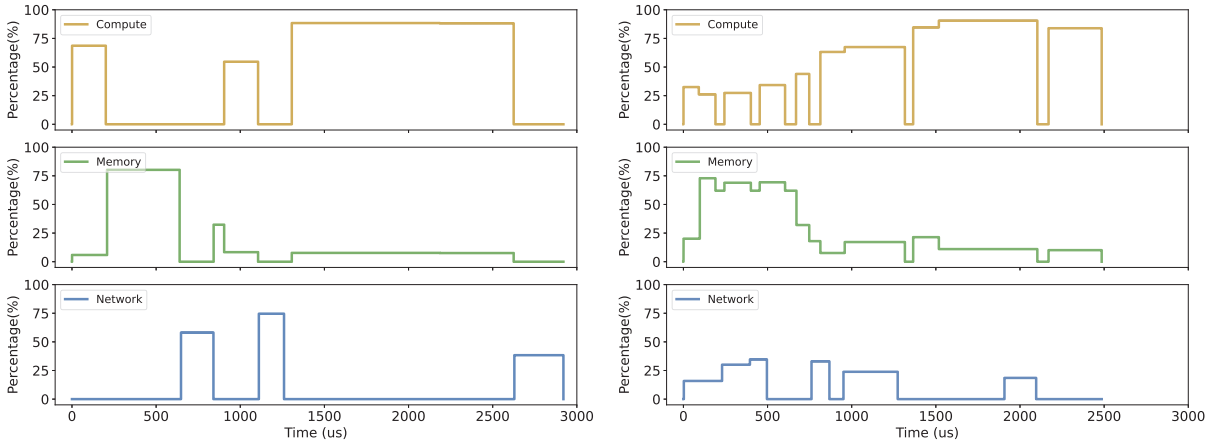
6.6 Performance on Other LLMs

We evaluate the throughput of NanoFlow instances for other LLMs with a constant length of input 1024 and output 512. All evaluations are done on 8×A100 80GB SXM GPUs, except for LLaMA-3-8B, which runs on a single A100 80GB SXM. The results are summarized in Figure 11. We find that NanoFlow improves throughput to between 50% and 72% of optimal throughput for these models, surpassing vLLM.

7 Related Work

LLM serving optimizations. Prior works have investigated optimizations for improving the serving throughput at different granularities. Orca [53] proposes request-level continuous batching, which refills the on-the-fly batch to maximize batch size at the granularity of an iteration. DistServe [59] and Splitwise [32] explore phase-level scheduling, which disaggregates the prefill and decode phases into different clusters. DeepSpeed-FastGen [13] and Sarathi-Serve [2] propose a chunked prefill policy, which splits a prefill request into multiple smaller chunks and batches decode and prefill requests together to improve overall utilization. However, these works do not perform scheduling at the granularity of operations. NanoFlow exploits intra-device parallelism, which features fine-grained resource management. Other works tackle the inference inefficiency by optimizing specific operations, including PageAttention [17], quantization approaches [19, 44, 55], etc. NanoFlow can easily adopt new operation implementations by profiling their performances and interference characteristics and integrating them into auto-search.

Operation-level parallelism. Some works focus on improving the efficiency of DNN workloads with operation-level optimizations. For example, Rammer [20] explores intra-operation parallelism by remapping the operations into different functional units. Unity [50] investigates the combination



(a) Non-overlap pipeline resource usage

(b) NanoFlow resource usage

Figure 10: The non-overlapping pipeline and NanoFlow's resource usage during inference of a single layer. NanoFlow achieves high compute usage across the whole pipeline by simultaneously utilizing memory and network bandwidth.

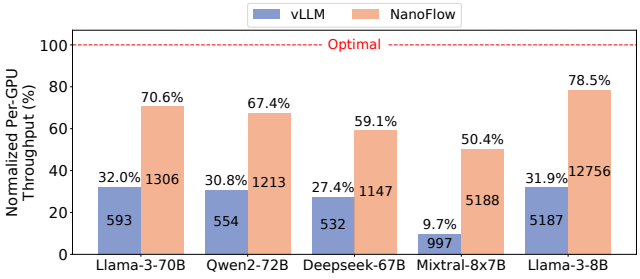


Figure 11: Performance of NanoFlow instances in terms of tokens per second per GPU for other models. Compared with vLLM, NanoFlow significantly increases throughput. NanoFlow achieves up to 78.5% of optimal throughput.

of parallelism and equivalent algebraic transformation of operations. However, these works do not consider the different resource demands of operations. ASPEN [30] and Welder [37] break the boundary between operations by constructing and compiling a tile-level data graph. Since tiles follow sequential dependencies, parallelism is still limited. Moreover, prior works require recompilation from scratch, which requires significant manual work. Compared with them, NanoFlow creates more overlapping opportunities through nano-batching.

8 Conclusion

In this work, we analyzed the characteristics of different operations in LLM serving and revealed the compute-bound nature of modern LLM serving workloads. We identified that compute is underutilized within a single GPU due to the sequential execution of compute-bound, memory-bound, and network-bound operations, leading to sub-optimal throughput. To address this, we proposed NanoFlow, a novel end-to-end LLM serving system that overlaps operations with heterogeneous resource usage through intra-device parallelism. NanoFlow automatically constructs a pipeline that overlaps nano-operations, with an auto-search that adapts to various LLM models. Our experiments show that NanoFlow achieves 1.91× throughput improvement over the state-of-the-art systems.

References

[1] Sharegpt. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2023.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. arXiv preprint arXiv:2403.02310, 2024.

[3] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. arXiv preprint arXiv:2308.16369, 2023.

[4] Mistral AI. Mistral nemo. https://mistral.ai/news/mistral-nemo/, 2024.

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

4901, Singapore, December 2023. Association for Computational Linguistics.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

[8] Marc Brysbaert. How many words do we read per minute? a review and meta-analysis of reading rate. Journal of Memory and Language, 109:104047, 2019.

[9] Alibaba Cloud. Alibaba cloud's qwen2 with enhanced capabilities tops llm leaderboard. https://www.alibabacloud.com/blog/alibaba-cloud's-qwen2-with-enhanced-capabilities-tops-llm-leaderboard_601268/, 2024.

[10] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017.

[11] Erin Griffith. The desperate hunt for the a.i. boom's most indispensable prize. Technical report, The New York Times, 2023.

[12] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: Modeling and optimizing training of large-scale dynamic pre-trained models. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '22, page 120–134, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. arXiv preprint arXiv:2401.08671, 2024.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[15] Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. Intelligent router for llm workloads: Improving performance through workload-aware load balancing, 2025.

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

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

[18] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 663–679, 2023.

[19] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving, 2024.

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

[21] Yusuf Mehdi. Reinventing search with a new ai-powered microsoft bing and edge, your copilot for the web, May 2023.

[22] Meta. Build the future of ai with meta llama 3. https://llama.meta.com/llama3/, 2024.

[23] Microsoft. Deepspeed: Fastgen. https://github.com/microsoft/DeepSpeed/tree/429bc5c/blogs/deepspeed-fastgen, 2024. Commit ID: 429bc5c, Accessed: 2024-12-09.

[24] NVIDIA. Nvidia dgx platform. https://www.nvidia.com/en-us/data-center/dgx-platform/, 2024.

[25] NVIDIA. Nvlink and nvlink switch. https://www.nvidia.com/en-us/data-center/nvlink/, 2024.

[26] NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2024.

[27] NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM/tree/5955b8a, 2024. Commit ID: 5955b8a, Accessed: 2024-12-09.

[28] OpenAI. Introducing chatgpt, 2023.

[29] OpenAI. Gpt-4 technical report, 2024.

[30] Jongseok Park, Kyungmin Bin, Gibum Park, Sangtae Ha, and Kyunghan Lee. Aspen: Breaking operator barriers for efficient parallelization of deep neural networks. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, Advances in Neural Information Processing Systems, volume 36, pages 68625–68638. Curran Associates, Inc., 2023.

[31] Dylan Patel and Afzal Ahmad. The inference cost of search disruption – large language model cost analysis, February 2023.

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[33] Archit Patke, Dhemath Reddy, Saurabh Jha, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hierarchical autoscaling for large language model serving with chiron, 2025.

[34] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. Proceedings of Machine Learning and Systems, 5, 2023.

[35] Emma Roth. Chatgpt's weekly users have doubled in less than a year, 2024. Accessed: 2024-12-03.

[36] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In International Conference on Machine Learning, pages 31094–31116. PMLR, 2023.

[37] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 701–718, Boston, MA, July 2023. USENIX Association.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[40] Shubham Singh. ChatGPT Statistics (AUG 2024) - Users Growth Data — demandsage.com. https://www.demandsage.com/chatgpt-statistics/, 2024. [Accessed 14-08-2024].

[41] Jared Spataro. Introducing microsoft 365 copilot – your copilot for work, May 2023.

[42] Vikranth Srivatsa, Dongming Li, Yiying Zhang, and Reyna Abhyankar. MLSys @ WukLab - Can Scheduling Overhead Dominate LLM Inference Performance? A Study of CPU Scheduling Overhead on Two Popular LLM Inference Systems — mlsys.wuklab.io. https://mlsys.wuklab.io/posts/scheduling_overhead/, 2024. [Accessed 25-10-2024].

[43] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24, page 1075–1092, New York, NY, USA, 2024. Association for Computing Machinery.

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

[45] The AIBrix Team, Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, Yifei Zhang, Yiqing Zhu, Shuowei Jin, Gangmuk Lim, Binbin Chen, Zuzhi Chen, Xiao Liu, Xin Chen, Kante Yin, Chak-Pong Chung, Chenyu Jiang, Yicheng Lu, Jianjun Chen, Caixue Lin, Wu Xiang, Rui Shi, and Liguang Xie. Aibrix: Towards scalable, cost-effective large language model inference infrastructure, 2025.

[46] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[47] CHENG TING-FANG. Tsmc sees ai chip output constraints lasting 1.5 years. Technical report, Nikkei Asia, 2023.

[48] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288, 2023.

[50] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 267–284, Carlsbad, CA, July 2022. USENIX Association.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[52] vllm project. vllm. https://github.com/vllm-project/vllm/tree/38c4b7e, 2024. Commit ID: 38c4b7e, Accessed: 2024-12-09.

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

[54] Liangyu Zhao, Saeed Maleki, Aashaka Shah, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. Forestcoll: Efficient collective communications on heterogeneous network fabrics, 2024.

[55] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. arXiv preprint arXiv:2310.19102, 2023.

[56] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

[57] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 559–578, 2022.

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

[59] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. arXiv preprint arXiv:2401.09670, 2024.

PipeThreader: Software-Defined Pipelining for Efficient DNN Execution

Yu Cheng[†*] Lei Wang[†] Yining Shi[†] Yuqing Xia[◇] Lingxiao Ma[◇] Jilong Xue[◇]
Yang Wang[◇] Zhiwen Mo[‡◇] Feiyang Chen[¶◇] Fan Yang[◇] Mao Yang[◇] Zhi Yang[†]
[†]*School of Computer Science, Peking University* [◇]*Microsoft Research*
[‡]*Imperial College London* [¶]*Shanghai Jiao Tong University*

Abstract

To effectively utilize heterogeneous specialized hardware units in modern GPUs, such as TensorCores and Tensor Memory Accelerators, this paper introduces PipeThreader, a new DNN compiler. PipeThreader proposes shifting scheduling functionality from hardware to software so as to enable more efficient and sophisticated computation pipelining with minimal manual effort. This is achieved through sTask-graph, a new DNN computation abstraction, a hierarchical hardware abstraction that captures the capabilities of specialized units, and new scheduling primitives. As a result, PipeThreader can discover efficient pipeline scheduling for well-studied DNN architectures like FlashAttention, achieving comparable or even superior performance. Additionally, it can uncover novel pipeline schemes for emerging models like Mamba2, delivering significantly better performance compared to state-of-the-art hand-crafted implementations. The code is open-sourced at https://github.com/tile-ai/tilelang.

1 Introduction

The increasing size of deep neural networks (DNNs) imposes significant computation and memory pressure on modern AI accelerators, such as GPUs. To meet the growing compute demands, hardware vendors have introduced specialized heterogeneous hardware units, such as TensorCores [14–16, 20–22] and Tensor Memory Accelerators (TMA) [22]. Meanwhile, software developers tend to fuse multiple DNN operators, a compute-intensive unit such as matrix multiplication, into one single GPU kernel to maximize data reuse so as to alleviate memory pressure [27, 47].

However, these emerging hardware and software trends present new challenges for efficient DNN execution. First, maximizing utilization of the specialized hardware units requires careful scheduling of the DNN computation pipeline. Previously, GPUs relied on hardware schedulers for thread execution, amortizing potential stalls within individual pipelines through *massive* concurrent threads. However, this approach is no longer effective, as the number of available concurrent threads has decreased significantly due to the inherently larger tensor granularity required by specialized units to achieve higher computation density. Second, operator fusion introduces deeper computation pipelines. Hardware schedulers struggle to comprehend such sophisticated pipelines, making effective scheduling challenging. As a result, state-of-the-art DNN kernels, such as FlashAttention [26], are hand-crafted to carefully organize the execution pipeline. However, this approach is hard to generalize. New hand-crafted implementations are required for every GPU type, e.g., the NVIDIA Hopper GPU or AMD GPU [46], emerging DNN models, e.g., Mamba [31], and even new tensor shapes of DNN models, e.g., FlashAttention2/3 [26, 46].

Given the growing complexity of software and hardware pipelines, coupled with the inherent limitations of hardware scheduling, this paper proposes PipeThreader, a DNN compiler that supports *software-defined* pipelining for efficient DNN execution on modern GPU architectures with specialized heterogeneous hardware units. PipeThreader abstracts DNN computation as an sTask-graph. Each node in the graph denotes an sTask, a fine-grained task that can be scheduled to a specialized unit; and directed edges in the graph specify dependencies across sTasks. sTasks compute on *tiles*, which are portions of a tensor. The concept of tiling is widely adopted by modern DNN compilers [11, 23, 43, 49, 53, 54, 56]. Using the sTask-graph, PipeThreader can extract sophisticated computation pipelines that are well-suited for execution on specialized hardware units.

To expose fine-grained scheduling capabilities of hardware, PipeThreader abstracts the GPU device as hierarchical hardware units, including the virtualized homogeneous parallel execution units (i.e., EU) and the heterogeneous specialized units within each EU (i.e., sEU). An sTask can then be scheduled using three primitives, `append`, `wait`, and `propagate`, thereby enabling sophisticated pipelining through software. PipeThreader also incorporates a two-layer scheduling policy to find effective scheduling schemes, given the new optimization space defined by an sTask-graph and the new exposed hardware capabilities.

We have implemented PipeThreader with 8.5k lines of C++ and Python code, based on open-source DNN compilers: TVM [19] and Ladder [51]. Evaluation shows that PipeThreader can discover the pipelining scheme resembling that of FlashAttention in new hardware like NVIDIA H100 or AMD MI300X GPUs, achieving comparable or better perfor-

*Part of work is done during the internship at Microsoft Research.

mance without manual implementations. PipeThreader can also find effective pipeline scheduling for emerging models like Mamba2 [28], producing significantly better performance than the state-of-the-art hand-crafted implementation. We believe that PipeThreader is a further step toward defining a complete scope of tile-based software/hardware abstraction, which well aligns with modern tile-based programming efforts (e.g., Triton [49], CuTe abstraction from CUTLASS [23], etc.) and the evolution of new hardware architectures.

2 Motivation

The growing hardware and software complexity. The rapid growth of large DNN models, particularly large language models (LLMs) [44], has driven hardware vendors to develop specialized heterogeneous hardware units, such as TensorCores and Tensor Memory Accelerators (TMA), to meet the escalating computational demands. Meanwhile, sophisticated operator fusion techniques like FlashAttention [27] are increasingly employed to reduce memory overhead and maximize data locality (details in §3.3). While these trends enhance computational density and efficiency, they also introduce significant challenges in scheduling and execution, particularly on modern GPUs with heterogeneous units.

Under-utilization in traditional data-parallel GPU execution. Traditional GPU programming models like CUDA [3] dispatch threadblocks across Streaming Multiprocessors (SMs), treating each SM as a uniform, isolated execution unit. This abstraction assumes all SMs are interchangeable and hides their internal structure, which worked well for earlier architectures like NVIDIA V100 [20]. However, modern GPUs like the NVIDIA H100 [22] integrate heterogeneous components within each SM, including TensorCores for matrix operations, CUDA cores for general-purpose computation, and TMAs for memory movement. These components have distinct roles and execution characteristics. Uniformly dispatching threadblocks without awareness of these internal differences leads to inefficient resource usage. To fully exploit such architectures, it is crucial to be aware of the heterogeneous units within each SM and to coordinate task placement, scheduling, and pipelining accordingly. Without this level of control, significant performance remains untapped.

Figure 1 reports per-unit utilization on H100 across different implementations of MatMul, FlashAttention, and Mamba2. For MatMul, the TensorCore utilization is only 40% due to the bottleneck on memory movement without the pipelined execution, while expert-optimized cuBLAS achieves 97% TensorCore utilization. FlashAttention-3 [46] improves the utilization of TensorCore from 40% to 72% with manual optimizations compared to Triton-based FlashAttention2 [26]. Notably, evolving from FlashAttention-2 to FlashAttention-3 took nearly one year. However, the emerging manually optimized Mamba2 [28] still underutilizes the units with only

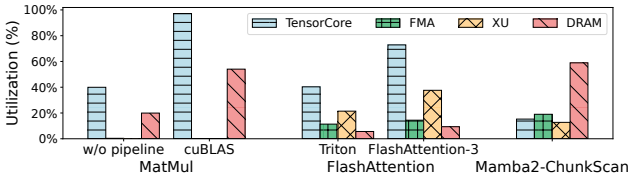


Figure 1: Per-unit hardware utilization on NVIDIA H100 for different implementations of representative AI workloads, including MatMul, FlashAttention, and Mamba2's ChunkScan. Each bar shows the utilization of individual hardware units for a specific workload implementation. Note that FMA and XU units are not used in MatMul.

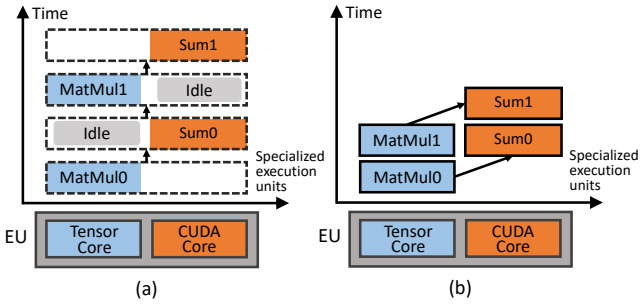


Figure 2: An illustration of (a) inefficient scheduling in existing approaches; (b) optimized scheduling that leverages pipelined execution across specialized execution units.

15% TensorCore utilization. Therefore, it is challenging to fully utilize the modern hardware for emerging DNN models.

Manually managing pipelined execution in kernels is notoriously challenging due to the vast and hardware-sensitive design space. Developers must carefully balance tile size and pipeline depth while adhering to tight on-chip resource constraints. These challenges are further amplified by architectural variability, including differences in memory hierarchies and specialized compute units. As manual reasoning quickly becomes intractable, automated inference and scheduling become essential for achieving both performance and portability. However, existing compilers such as TVM [19] and Triton [49] lack explicit mechanisms to express pipelined tile execution. By abstracting away low-level control, they limit the developer's ability to specify execution order, resource allocation, and compute-communication overlap—hindering the exploitation of full performance potential. Realizing efficient pipelined tile execution requires a new compiler that can systematically explore this design space, generate optimized schedules, and adapt across diverse hardware platforms.

Observations and opportunities. In light of these facts, we observe a unique opportunity to address the issue. Given that the new hardware units process data at a large granularity, such as tensor tiles, previous works [11, 23, 43, 49, 53, 54, 56] have demonstrated that tile-level execution can be efficiently scheduled at the software layer due to its deterministic performance at the tile level. By leveraging this trend, we advocate

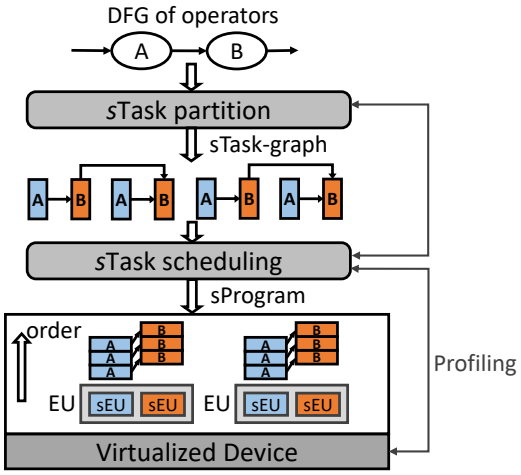


Figure 3: System overview of PipeThreader.

```
1  class sTaskGraph {
2      list<sTask> nodes;
3      list<pair<sTask, sTask>> edges;
4  };
5  class sTask {
6      size_t sTask_id;
7      TensorExpr expr;
8      TileShape shape;
9      sEUType target_sEU;
10 };
11 class vDevice {list<EU> EUs;};
12 class EU {list<sEU> sEUs;};
13 class sEU {
14     bool is_async;
15     void Execute(sTask s);
16 };
```

Figure 4: The abstraction of sTask and sEU.

shifting pipeline scheduling from implicit hardware behavior to explicit software control. Here, pipeline scheduling refers not to low-level thread or warp dispatching by the GPU, but to the software-guided mapping of tile-level operations onto specialized units such as TensorCores, CUDA cores, or TMAs within each SM.

Figure 2(a) shows the execution of fused MatMul-Sum where MatMul is executed on TensorCore and Sum is executed on CUDA core. Due to the homogeneous abstraction in existing approaches, the scheduling serializes the execution despite the inherent parallelism between TensorCores and CUDA cores, leading to inefficiencies. In contrast, Figure 2(b) demonstrates the optimized scheduling that leverages specialized execution units, enabling pipelined execution and fully utilizing the heterogeneous hardware.

Unfortunately, neither the existing DNN model representations nor the hardware interfaces in existing GPUs explicitly expose the scheduling capabilities required for tile-level pipeline execution.

3 PipeThreader Abstraction

The observations in §2 motivate PipeThreader, a DNN compiler framework that integrates tile-based data parallelism with pipelined scheduling. Figure 3 provides an overview of the system. State-of-the-art DNN compilers (e.g., Triton [49], Roller [56], Welder [47]) abstract hardware accelerators as collections of homogeneous execution units (EUs) for SPMD-style parallelism. This approach overlooks the hardware heterogeneity inherent in modern GPUs. For example, TensorCores and CUDA cores within an SM are optimized for different workloads, a diversity that existing compilers fail to utilize. To address these limitations, PipeThreader introduces two key abstractions: *specialized tasks* (*sTasks*) and *specialized execution units* (*sEUs*). Starting with a data flow graph (DFG) of operators as input, PipeThreader converts these operators into sTasks, which are designed to exploit the heterogeneous capabilities of sEUs, enabling MPMD-style parallelism. Details of sTasks and sEUs are discussed in §3.1. sTasks preserve the data dependencies of the original DFG at task granularity, forming an *sTask-graph*. PipeThreader organizes the mapping of the sTask-graph to sEUs into an *sTask-program*, a structured representation for execution. With the *sTask-program* abstraction, PipeThreader opens up a new search space. Details of the sTask-graph, sTask-program, and search space are discussed in §3.2.

3.1 Specialized Tasks and Execution Units

sTask. PipeThreader introduces sTasks (short for specialized tasks) as the basic computation unit in an operator to be executed on a specific execution unit (sEU) of the accelerator device. The concept of sTask naturally aligns with the heterogeneous, specialized processors of modern DNN accelerators, e.g., TMA, CUDA cores, and TensorCores in H100 GPU. To maximize efficiency, the computation on such an accelerator needs to be divided into multiple parallel (heterogeneous) tasks for each type of specialized processor. Each type of these parallel tasks can be represented by an sTask, thereby exposing the potential task parallelism not only to specialized processors of the underlying hardware but also to the PipeThreader compiler.

As shown in Figure 4, an sTask processes a data tile sliced from the input tensors and produces a data tile in the output tensors, with its computation described by an index-based tensor expression. The `shape` (line 8) of an sTask is defined along each loop axis of the tensor expression `expr` (line 7). Additionally, the `target_sEU` (line 9) attribute specifies the type of specialized units on which the sTask can execute. In contrast, traditional tile-based tasks are not explicitly categorized, limiting their ability to exploit the parallelism of specialized processors. As illustrated in Figure 5, on a specific EU, a traditional MatMul-Sum task (a fused operation in FlashAttention) would sequentially compute a $[2 \times 2]$ data tile from A and a $[2 \times 2 \times 2]$ data tile from B, producing a $[2 \times 2]$ output tile for C. PipeThreader introduces two types of sTasks: `mma` sTasks executing matrix multiply-accumulate on TensorCores, and `Sum` sTasks running on CUDA cores.

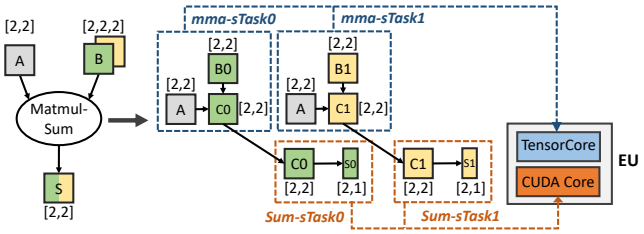


Figure 5: sTask MatMul-Sum on NVIDIA GPU.

This enables pipelining, as the second `mma` sTask, multiplying A with the partition $B1$ to produce $C1$ on TensorCores, can overlap with the first `Sum` sTask, reducing $C0$ to $S0$ on CUDA cores. This pipelined execution allows task parallelism, significantly improving hardware utilization.

sEU. Modern accelerators lack interfaces to map an sTask to a specific execution unit. To address this, PipeThreader explicitly exposes the execution units within GPUs and abstracts them as a hierarchical execution array, capturing both parallelism and the capability to support data-dependent execution with pipelining.

As illustrated in Figure 4, the abstracted device consists of multiple parallel execution units (EUs) (line 11), each containing several heterogeneous *specialized execution units* (*sEUs*) (line 12). These sEUs serve as the hardware foundation for PipeThreader to effectively schedule data-dependent tasks with pipelining. For instance, on a modern H100 GPU, a Streaming Multiprocessor (SM) is an EU that includes Tensor Memory Accelerators (TMAs) for `load` sTasks and TensorCores for `mma` sTasks. An sEU executes a given sTask using the `Execute` interface (line 15). The `is_async` attribute (line 14) specifies whether the sEU operates synchronously (e.g., CUDA cores) or asynchronously (e.g., TMAs). An asynchronous sEU can be executed concurrently with either asynchronous or synchronous sEUs.

3.2 From sTask-Graph to sProgram

To execute a DNN, PipeThreader transforms the input DFG into a specialized representation tailored to modern heterogeneous hardware. This process involves two key steps: constructing an *sTask-graph* that captures computation and dependency, and mapping this graph to specialized execution units (*s*EUs) as an *sTask-program* (*sProgram*), which orchestrates efficient execution.

sTask-Graph. As illustrated in Figure 3, operators from the input DFG are converted into *sTasks* via `sTask-partition`, forming an *sTask-graph*. This graph preserves the computation and data dependencies of the original DFG, with nodes representing sTasks and edges capturing their fine-grained dependencies at the task level. The `sTask-partition` process partitions each operator by configuring the `TileShape` (i.e., Map<Axis, Dim>) of sTasks, specifying their divisible dimensions and sizes. Traditional compilers primarily focus on spatial partitioning to achieve data parallelism. PipeThreader extends to support both spatial and reduction partitioning, unlocking new opportunities for pipelined execution. This flexibility allows PipeThreader to generate diverse sTask-graphs based on different partitioning strategies, enabling more versatile and efficient execution planning.

sProgram. Given the sTask-graph, PipeThreader maps it onto the hardware's sEUs in the form of an *sTask-program* (*sProgram*). The sProgram is a two-dimensional array, `sProg[sEU][order]`, where each entry specifies the assignment of an sTask to a particular sEU and its execution order. This structured representation facilitates efficient scheduling and execution of tasks. To maintain correct execution order for dependent sTasks, PipeThreader introduces *barrier-sTasks*, which synchronize execution by referencing a list of sTasks identified by `<EU_id, sEU_id, order>` in the program. A barrier-sTask waits for the completion of all referenced sTasks before proceeding.

Search Space. The search space of PipeThreader is structured as the set of sPrograms, where each is a two-dimensional array `sProg[sEU][order]` defining tiling size and execution order (with synchronization barriers) for each sTask in the graph. There might be many combinations of sTask ordering and tiling sizes. PipeThreader's search space includes all valid sPrograms where operators can be executed while respecting data dependency. For example, FlashAttention's search space includes 37,440 valid sPrograms. For complex fused operators, sTask scheduling often contributes more significantly to the search space. With more types of sTasks, there might be many valid sPrograms of different sTask execution orders. For example, in FlashAttention, there are 36 sTask size (i.e., tiling) configurations, but 1,040 sTask ordering configurations for each tiling size.

3.3 Running Examples

Mamba2. Mamba is a popular DNN model that processes sequences using chunk-wise scanning with a linear attention mechanism. Its linear attention comprises multiple modules; here, we illustrate its key ChunkScan operator.

Frontend. For the ChunkScan function, PipeThreader takes a simple IR represented in Figure 6(a) as input. It multiplies *cb* by the product of the exponential of *dA* and *dt* (line 6), then accumulates the result of the matrix multiplication of *cb* and *x* into *acc_o* (line 8). PipeThreader treats `load_cb` (line 3), `load_dA` (line 4), `load_dt` (line 5), `exp` (line 6), `load_x` (line 7), and `mma` (line 8) as individual sTasks.

sTask-graph. PipeThreader constructs a corresponding sTask-graph based on their dependencies. The sTask-graph can be partitioned along both spatial and reduction dimensions. Spatial partitioning (i.e., batch size) splits the graph into smaller subgraphs distributed across EUs for tile-based data parallelism. Reduction partitioning (i.e., sequence length)

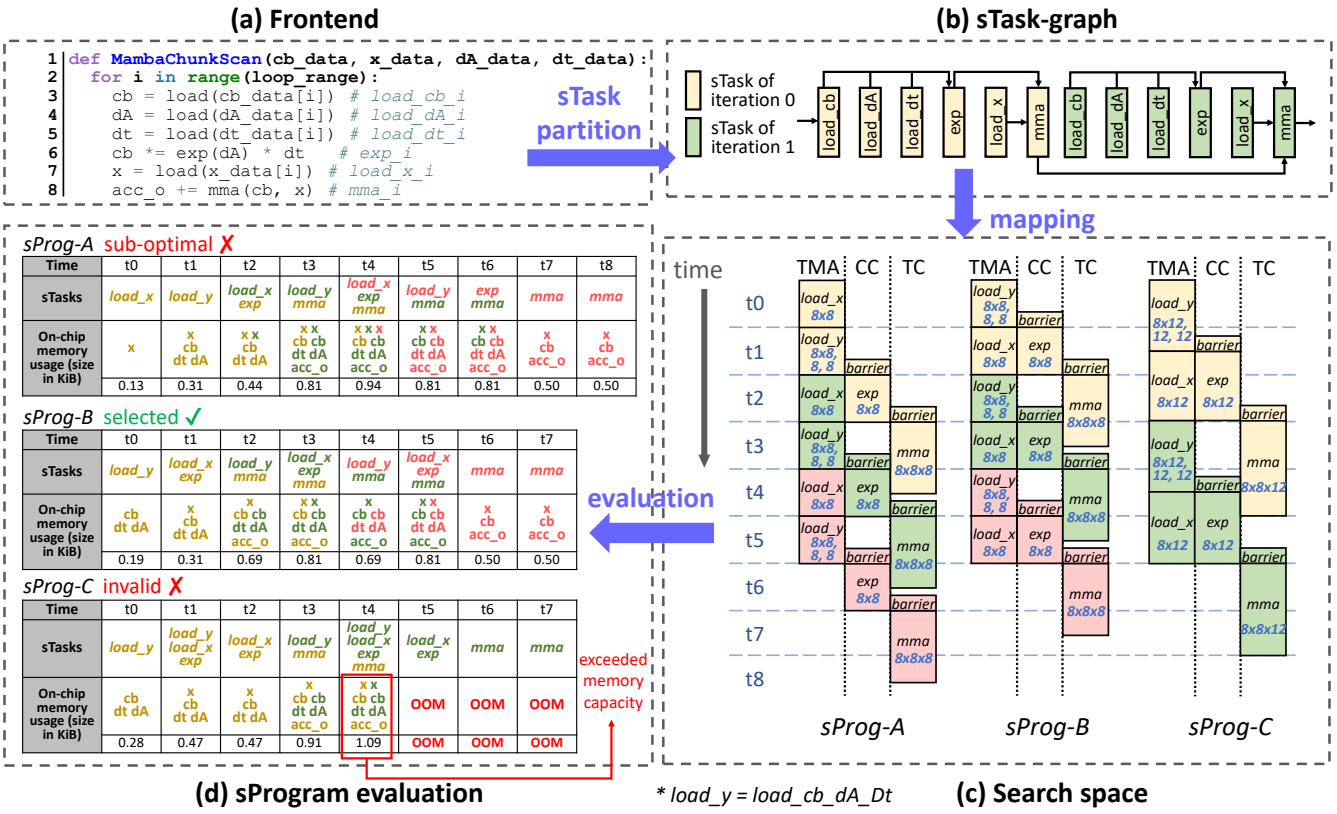


Figure 6: Running example of Mamba2-ChunkScan. (a) shows the user-facing frontend. (b) presents the sTask-graph constructed from (a), with colors indicating different iterations. (c) illustrates various sPrograms derived from the sTask-graph in (b), and (d) compares their evaluation to identify the most efficient sProgram.

creates finer-grained sTasks within an EU, exposing opportunities for pipelined execution. Given *acc_o* (line 8 in Figure 6(a)) of size (M,N) and X of size (K,N), PipeThreader partitions the spatial dimensions (M,N) as usual, assigning each sTask a tile of (m,n). It also partitions the reduction dimension (K) into *loop_range* iterations, allowing computations to overlap across iterations[1]. Figure 6(b) shows the sTask-graph derived from reduction-dimension partitioning.

sPrograms. Given the sTask-graph, PipeThreader can have different choices of mapping sTasks to sEUs in the form of multiple *sPrograms*. Figure 6(c) illustrates three sPrograms derived from the sTask-graph. In sProg-A, `load_x` is scheduled before other load sTasks, whereas sProg-B and sProg-C schedule these operations in reverse order. Compared to sProg-B, sProg-C employs a larger tiling size.

Evaluation. Figure 6(d) illustrates the evaluation of the three sPrograms. The table shows the sTasks running at each time step, along with the corresponding on-chip memory usage. Here, we assume an on-chip memory capacity of 1 KiB. In sProg-A, `load_x` is scheduled earlier, but since the `exp` sTask depends on the completion of `load_cb_dA_dt`, this results in delayed scheduling of `exp`, leading to lower overall efficiency compared to sProg-B. Although sProg-C employs larger sTasks, this increases the on-chip memory usage of its workspace. We observe that at time step t_4, the workspace exceeds the available on-chip memory capacity, rendering this sProgram invalid. Consequently, sProg-B is selected as the final scheduling strategy.

PipeThreader's task partitioning follows the principles of tiling but elevates new *reduction tiling* as a key optimization strategy. Traditionally, tiling prioritizes spatial partitioning for data reuse, while reduction tiling is often treated as secondary. PipeThreader, however, actively exploits reduction tiling to enable pipelining, improving execution efficiency. This ensures efficient pipelined execution while maintaining data reuse. By making reduction tiling a first-class optimization, PipeThreader unlocks performance improvements beyond conventional tiling strategies, particularly in pipeline-heavy workloads. Also, traditional tiling strategies tend to use larger tile sizes, which may conflict with increasing pipeline parallelism due to both demanding on-chip memory. PipeThreader further makes the trade-off between tiling and pipelining.

FlashAttention-3. FlashAttention is an efficient implementation of the original full attention mechanism, in which the input nodes of the DNN operator graph are three tensors: $Q, K,$

[1]Note here the sTask-graph does not fully unroll all the loops, it models iteration structures via scheduling a window of `MAX_STREAM` iterations.

```
1 def FlashAttention(q_data, k_data, v_data):
2     q = load(q_data)
3     for i in range(loop_range):
4         k = load(k_data[i]) # load_k_i
5         acc_s = mma(q, k)   # mma_qk_i
6         P = softmax(acc_s)  # softmax_i
7         v = load(v_data[i]) # load_v_i
8         acc_o = rescale(acc_o, param) # rescale_i
9         acc_o += mma(P, v)  # mma_pv_i
```

Figure 7: Pseudocode of the FlashAttention sTask-graph.

and V. First, a matrix multiplication `MatMulQK` is performed to compute $acc_s = QK^{\mathrm{T}}$. Next, acc_s is passed through a `Softmax` operation to produce P. Finally, P and V are used as inputs for the second matrix multiplication `MatMulPV` to compute the output O.

In FlashAttention, these three operators are fused into a single kernel. PipeThreader annotates the pattern to derive dependent sTasks, forming an sTask-graph (Figure 7). After partitioning, PipeThreader assigns `load_k` (line 4) and `load_v` (line 7) to TMA, `mma_qk` (line 5) and `mma_pv` (line 9) to TensorCores, and `softmax` (line 6) and `rescale` (line 8) to CUDA cores. PipeThreader applies its two-level scheduling policy to explore the space and generates an sProgram where sTasks execute in parallel on their respective sEUs, leveraging the asynchrony and heterogeneity of sEUs. It is worth noting that our scheduling space includes the pipeline plans of the latest FlashAttention-3 [46].

4 PipeThreader Scheduling

The sProgram abstraction opens up a large optimization space. PipeThreader aims to generate high-quality sPrograms in this space. To this end, PipeThreader separates the scheduling mechanism from its policy. On the mechanism side, it provides two capabilities: (1) Scheduling interfaces for a policy to generate a sProgram. (2) A profiler to supply profiling information requested by a scheduling policy. On the policy side, PipeThreader provides a two-layer policy that balances tiling and pipeline parallelism. This simple policy can already outperform the state-of-the-art, sometimes significantly. We believe this mechanism lays a foundation for future research into more sophisticated policies that further exploit the optimization space exposed by sPrograms.

4.1 Scheduling Interfaces

PipeThreader provides three interfaces to generate a high-quality sProgram in the new space, as shown in Figure 8. The `Append` interface assigns a specific sTask to a particular *s*EU within an EU. The `Wait` interface allows an sTask `s` to wait for sTasks in `list<sTask_uid>` to complete, which implicitly appends a barrier-sTask right before the sTask `s`. The above interfaces allow for explicit control of the placement and execution order of sTasks across sEUs (i.e., sProgram) to explore parallelization space.

```
1 void Append(sTask s, <EU u, sEU v>);
2 void Wait(sTask_id s, list<sTask_id> t);
3 void Propagate(sTaskGraph g, TileShape shape);
```

Figure 8: The scheduling interfaces.

PipeThreader provides a `Propagate` interface to explore the sTask partition space by automatically inferring the `TileShape` of each sTask in an sTask-graph. Starting from the output tile shape of the final sTask, `Propagate` performs a chain of shape inferences backward through the graph, determining the dependent input regions for each sTask based on its tensor expression and output tile shape. For example, if $[4 \times 128]$ output tile shape is required for a `Softmax` sTask, `Propagate` deduces that its input tile shape must also be $[4 \times 128]$. Treating this as the output tile shape of a preceding `mma` sTask, the input tiles are inferred as $[4 \times k]$ and $[k \times 128]$, where k is the reduction size.

4.2 Scheduling Policy

Our scheduling policy is inspired by the two-layer hardware abstraction described in Figure 3, where homogeneous EUs enable SPMD-style parallelism, and heterogeneous sEUs within each EU support MPMD-style parallelism. Figure 9 outlines the two-level scheduling algorithm used in PipeThreader. At the inter-EU level, the policy minimizes latency by partitioning the model into sTask-subgraphs, which are evenly distributed across EUs (lines 1-10). At the intra-EU level, the policy optimizes the execution cost of each sTask-subgraph on a given EU by constructing an efficient pipelining plan (lines 11-32). The inter-EU schedule is informed by execution cost estimates for each partition, as provided by the intra-EU schedule.

Initially, the policy represents each operator as one or more sTasks based on its computational stages (e.g., MatMul is split into `load` and `mma` sTasks). In the inter-EU pass, the scheduler enumerates different partitions of the output sTask in the function `GetsTaskPartitions` (line 2). For each sTask partition, the scheduler uses `Propagate` (line 4) to derive other sTask partitions across the graph and assigns sTasks evenly across EUs, leveraging their equivalent compute capability. This SPMD-style approach significantly reduces the complexity of inter-EU parallelism plans. The policy invokes the intra-EU pass to optimize the execution of assigned sTasks within an EU (i.e., an sTask-subgraph). During the intra-EU pass, the policy employs a greedy approach to schedule sTasks onto sEUs, iteratively performing the following steps until all sTasks assigned to the EU are scheduled: 1) selects the sTask t with the `endtime` earlier than the current time `cur_time` in `get_complete_sTask` (line 15); 2) identifies the set of ready sTasks whose predecessors have been scheduled (line 17-20) and dequeues the sTask u with the highest priority using `get_high_priority` (line 22); 3) appends the selected sTask to the sEU with `Append()` (line 23) and ensures sTask-level dependencies by invoking `Wait()` (line 25). We also call

```
1  Func InterEUScheduling(G:operator-graph, D:vDevice):
2      S = GetsTaskPartitions(out_op);
3      for out_tiles ∈ S.TileShape do
4          sTask_set = Propagate(G, out_tiles);
5          sTask_subgraph = Partition(sTask_set, D);
6          IntraEUScheduling(sTask_subgraph, EU, plan);
7          if Profile(plan).latency < best_latency
8              best_latency = Profile(plan).latency;
9              best_plan = plan;
10     return best_plan;
11 Func IntraEUScheduling(g:sTask-graph, EU, Plan):
12     scheduled_set =∅;
13     curr_time = 0;
14     while scheduled_set ≠ g.nodes do
15         t = get_complete_sTask(Plan, curr_time);
16         t.state = COMPLETE;
17         for n ∈ g.nodes \scheduled_set do
18             if all sTasks in Predecessor(n) are COMPLETE
19                 n.state = READY;
20                 ready_sTasks.add(n);
21         for ready_sTasks≠ ∅ do
22             u= get_high_priority(ready_sTasks, Plan, g);
23             P = Plan.Append(u, <EU, sEU>);
24             if check_valid(P)
25                 P.Wait(u, Predecessor(u).sTasks);
26                 if t ∉ Predecessor(u)
27                     P.Wait(u, t);
28                 u.endtime = max{sEU.time, curr_time} +
                     Profile(u, sEU);
29                 sEU.time = u.endtime;
30                 scheduled_set.add{u};
31                 Plan = P;
32         curr_time = curr_time + δ;
```

Figure 9: Scheduling algorithm.

`Wait(u, t)` to handle cases where the scheduling of u must wait for the completion of t to release memory (line 26-27). To improve pipeline efficiency, we prioritize scheduling asynchronous sTasks with *minimal dependencies on already scheduled tasks* and *high potential to unlock downstream sTasks*. As illustrated in Figure 6(c), scheduling `load_x` early delays the execution of `exp`, whereas scheduling `load_y` earlier enables `exp` to proceed sooner. As a result, our algorithm assigns a higher priority to `load_y`, naturally favoring the construction of `sProg-B` and `sProg-C` over `sProg-A`.

Increasing sTask overlap (i.e., pipeline parallelism) requires additional on-chip memory (e.g., shared memory and registers on GPUs) to buffer intermediate results between stages. However, this demand may conflict with the use of larger tiling sizes. Our method balances these competing demands through a joint search strategy guided by profiling feedback. To ensure memory feasibility, we invoke `check_valid` to verify (based on the current sProgram and profiler) whether a selected sTask fits within the memory constraints of the target `sEU` (line 24).

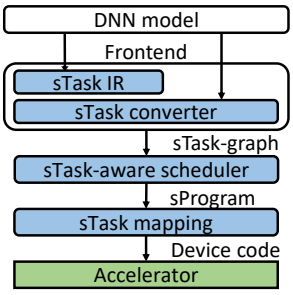


Figure 10: Implementation of PipeThreader.

Candidates that exceed the limit are skipped. For example, in Figure 6, `sProg-C` employs a larger tiling size, which exceeds the available on-chip memory. The `check_valid` step detects this violation and avoids generating such invalid schedules.

Profiler. PipeThreader introduces a profiler to guide efficient sProgram generation (Figure 9) in the search space. The profiler provides the following information of individual sTasks to generate a valid execution timeline of the sProgram: (1) the execution time of individual sTasks on specific sEUs, (2) the resource usage of sTasks, including local memory and register consumption, and (3) the overall execution time of the sProgram. The profiler automatically handles a new tensor expression by leveraging the code generation backend of existing compilers like TVM, and measures the execution time and resource usage of the device code of an isolated sTask. During scheduling, PipeThreader uses these profiling results to estimate when a task should be launched to maintain pipeline efficiency and minimize idle time. After completing sTask scheduling, the profiler also measures the performance of the entire generated schedule, providing the ground-truth latency. This profiling data informs the scheduling policy and guides the generation of efficient scheduling plans.

5 PipeThreader Implementation

PipeThreader is implemented with 8.5k lines of C++ and Python code, based on open-source DNN compilers: TVM [19] and Ladder [51]. Figure 10 summarizes the overall workflow of PipeThreader. The frontend produces the sTask-graph, which is then processed by the sTask-aware compiler (scheduler) to generate the sProgram. Finally, the mapping optimizer generates the device code for the sProgram.

5.1 Frontend

The PipeThreader frontend includes an sTask-IR for expressing the sTask-level DNN computations, and an sTask-converter to transform DNN model into sTask-graph.

sTask IR. The sTask intermediate representation (IR) provides the programmer and the compiler with a flexible way to express sTask-level computation that cannot be easily captured by the existing compiler IRs (e.g., expression-oriented

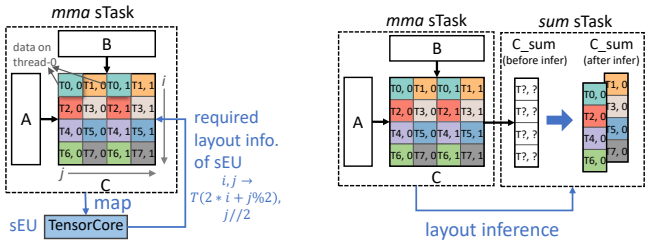
IR). The pseudocode in Figure 6(a) and Figure 7 can be viewed as a simplified form of the sTask IR. The pseudocode illustrates how complex deep learning kernels can be modeled as data-flow patterns, including memory operations (e.g., moving sTasks between DRAM and SRAM) and sequences of computations on sTasks.

sTask converter with sEU. The PipeThreader frontend can also convert the DNN model expressed by both sTask IR and ONNX graphs into an sTask-graph. In this process, we leverage Ladder [51], a state-of-the-art DNN compiler for operator fusion. The output of Ladder is a tile-graph, which is expressed in TVM's TIR as an intermediate representation. We annotate the `target_sEU` attribute of each tile-based task to transform it into sTasks based on sEU information. For instance, on the NVIDIA H100 GPU, we treat a Streaming Multiprocessor (SM) as an EU. Within each SM, the sEUs include the TensorCores for matrix multiply-accumulate `mma`, CUDA cores for general floating-point computations such as `reduce` and `parallel`, and the TMA for bulk memory `copy` operations between global and shared memory. These basic operations can be composed to represent data operations in most common deep learning kernels. For example, data movement is translated into a `parallel` operator which can represent arbitrary element-wise tile operations. Also, users can define customized functions to describe other sTasks.

Although programmers need to write a simple IR to generate kernels (e.g., Figure 7 for the FlashAttention kernel), they do not have to be aware of tasks, how to divide their graphs (or IRs) into sTasks, and which sTasks can run on which sEUs. PipeThreader can infer this information, for example, setting attributes such as tiling shape and target sEU in the sTask class in Figure 4. The sTask converter annotates each operation (e.g., `mma`) on which types of sEU to execute (i.e., `target_sEU` attribute). The scheduler then automatically determines how to divide it into sTasks (e.g., tiling shape) and which sTasks run on which sEUs (e.g., sEU assignment in sProgram). Therefore, PipeThreader can help reduce the tedious manual effort and strong domain expertise required for hand-crafted implementation. For the FlashAttention kernel, PipeThreader only needs 68 lines of Python code, as compared to 840 lines of CUDA kernel code from the hand-crafted implementation FlashAttention-3.

5.2 sTask Mapping on NVIDIA CUDA GPUs

For both TMA and TensorCore sEUs, the `is_async` attribute is set to true, as we can leverage the `cp.async.bulk` and `wgmma.mma_async` instructions. In contrast, the `is_async` attribute for CUDA cores is set to false because they do not support any asynchronous instructions. The fact that instructions for CUDA Cores and TensorCores can be dispatched simultaneously is officially confirmed by NVIDIA [8]. Interference can occur as both units utilize the same set of registers. We alleviate the potential interference by implementing double buffering on registers to overlap the execution of TensorCore and CUDA Core. We implement barrier-sTask using the `mbarrier` object in PTX [9]. To efficiently implement the `Execute` function in sEU, we decide how sTask operations and data map to different threads and physical memory through layout inference. We also use hardware-specific instructions (hardware intrinsics) to accelerate operations. PipeThreader does not require significant engineering effort to support different GPU models from the same vendor. When targeting different architectures such as A100 [21], H100 [22], or B100 [24], only lightweight updates to hardware-specific configurations, including sEU layouts, intrinsics, and resource limits, are needed. The core compilation and scheduling logic remains fully reusable.



(a) Align the sTask's layout with sEU requirements.

(b) Infer the layouts of connected sTasks on the sTask-subgraph.

Figure 11: An example of simplified layout inference for the `mma-sum` sTask-subgraph.

Layout inference. Efficient execution of sTasks on specialized execution units (sEUs) requires adherence to specific layout and thread-binding constraints. To address this, PipeThreader introduces a `Layout` object, which describes the data layout and thread-binding for sTasks. The `Layout` defines a mapping function and an iterator domain, specifying how logical data elements are translated to physical memory and, optionally, assigned to threads.

PipeThreader performs layout inference fully automatically, eliminating the need for manual specification. Figure 11a shows a simplified version of layout inference. The sTask `mma` is assigned to sEU TensorCore with strict layout constraints. Based on the layout constraints, we can derive the corresponding layout mapping function. Here, $\{T(m), n\}$ denotes the data element mapped to the n-th position of thread m.

In the sTask-graph, the layouts of connected sTasks must align to ensure compatibility. Using the layout requirements of specific sEUs, PipeThreader infers the `Layout` of sTasks and propagates these requirements throughout the graph. Conflicts between layouts are resolved through a priority-based inference algorithm, where high-priority sTasks, such as `mma`, dictate the layouts of dependent sTasks. For instance, in Figure 11b, the `mma` and `sum` sTasks are connected. Given that the layout of the `mma` sTask has already been determined, we can infer the layout of the `sum` sTask accordingly. In this example, the tensor `C_sum` needs to be duplicated.

Hardware intrinsic. For sTasks requiring bulk operations on sEUs, we lower the sTask into tile-level function templates. For example, matrix multiply-accumulate operations are lowered using the CUTLASS/CuTe templates, which integrate hardware-specific TensorCore intrinsics. Further instruction-level optimizations, such as register allocation, are delegated to low-level compilers like LLVM [38]. For NVIDIA H100, we apply Warp Specialization [17, 25] to optimize execution. This technique divides threads into producer and consumer warps, with each warp responsible for different pipeline stages. By allowing producer warps to release unused registers for reuse by consumer warps, Warp Specialization improves register allocation and efficiency. Based on the features of the TMA unit in H100, we assign the `load` sTasks, which copy data between global and shared memory, to producer warps. The remaining sTasks, such as `mma` and `Softmax`, are handled by consumer warps. Synchronization between producers and consumers is achieved using barrier-sTasks implemented with `mbarrier` to ensure correct data dependencies.

5.3 sTask Mapping on AMD ROCm GPUs

We also implement PipeThreader on MI300X [16], AMD's latest high-performance GPU. The MI300X GPU features parallel execution units known as compute units (CUs), analogous to NVIDIA's SMs. Each CU contains multiple sEUs, including MatrixCores for matrix multiply-accumulate, Arithmetic Logic Units (ALUs), and asynchronous copy units. Similar to CUDA GPUs, PipeThreader performs layout inference on ROCm GPUs to map sTask data to physical addresses and threads. Additionally, we explicitly utilize the `lgkmcnt` and `s_waitcnt` instructions to manage asynchronous barriers, enabling precise control over instruction dependencies and synchronization of memory operations.

6 Evaluation

In this section, we evaluate PipeThreader on both DNN microbenchmarks and end-to-end models by comparing with state-of-the-art DNN compilers, frameworks, and libraries to demonstrate the effectiveness of PipeThreader. We first summarize our findings: (1) PipeThreader can discover efficient scheduling for well-established DNN architectures (e.g., FlashAttention) and achieve performance comparable to or even surpassing state-of-the-art; (2) PipeThreader can uncover novel scheduling for emerging models (e.g., Mamba2), delivering significantly improved performance; (3) PipeThreader's abstraction and design are adaptable to hardware beyond NVIDIA GPUs (e.g., AMD MI300X), achieving notable performance gains.

6.1 Experimental Setup

Hardware platforms. We evaluate PipeThreader on both NVIDIA and AMD GPUs, as they are currently the most popular hardware platforms. Our evaluation includes two of the latest high-performance GPUs: the NVIDIA H100 (80GB) [22] and the AMD Instinct MI300X GPU (192GB) [16]. We use CUDA version 12.4 with the H100 GPU and ROCm version 6.1.0 with the MI300X GPU. Both GPUs are evaluated on the operating system Ubuntu 20.04.

Operator	Configuration	Model	Note
MatMul	M=4096, N=1024 K=4096	LLAMA3-8B	M0
MatMul	M=8192, N=14336, K=4096	LLAMA3-8B	M6
MatMul	M=8192, N=8192, K=28672	LLAMA3-70B	M15
Conv2D	D=(1,2048,7,7), K=(2048,512,1,1), S=1	ResNet-50	C0
Conv2D	D=(128,512,14,14), K=(512,512,3,3), S=2	ResNet-50	C10
Conv2D	D=(128,512,28,28), K=(512,128,1,1), S=1	ResNet-50	C14
$W_{FP4}A_{FP16}$ MatMul	M=64, N=1024, K=8192	LLAMA3-70B	DM0
$W_{FP4}A_{FP16}$ MatMul	M=128, N=28672, K=8192	LLAMA3-70B	DM6
FlashAttention	BS=1, SEQ=4096, causal=true	LLAMA3-8B	FA4
FlashAttention	BS=64, SEQ=4096, causal=false	LLAMA3-70B	FA19
FlashDecoding	BS=1, SEQ=1, causal=false	LLAMA3-8B	FD0
FlashDecoding	BS=1, SEQ=1, causal=false	LLAMA3-70B	FD2
ChunkScan	BS=1, SEQ=8192	Mamba2-1.3B	CC2
ChunkScan	BS=64, SEQ=16384	Mamba2-1.3B	CC7
ChunkState	BS=1, SEQ=8192	Mamba2-1.3B	CT2
ChunkState	BS=64, SEQ=16384	Mamba2-1.3B	CT7

Table 1: A subset of operator configurations in our microbenchmark.

DNN workloads. Our evaluation benchmark uses six typical DNN models, including LLAMA3-8B [29], LLAMA3-70B [29], Mamba2-1.3B [28], RetNet-65B [48], ResNet-50 [32], and UNet [45]. For large language models like LLAMA3-8B, LLAMA3-70B, and RetNet-65B, we perform tests using the (BS, SEQ) configurations of (1, 1), (32, 1), and (1, 4096); other models like ResNet-50 and UNet are evaluated with batch sizes of 1 and 128, which can comprehensively cover both online and offline inference scenarios. For Mamba, we evaluate BS=1 with sequence lengths of 1k, 2k, 4k, and 8k, and BS=32 or 128 with a sequence length of 1. These configurations are chosen because Mamba's primary advantage over transformers is its superior computational efficiency with long sequence lengths, and these settings represent the most commonly used scenarios. From each model, we choose the most frequent and expensive operations to construct our microbenchmark. Table 1 lists representative operators, their configurations, and corresponding abbreviations of each operator.

Baselines. We compare PipeThreader with DNN framework ONNXRuntime (v1.19.2) [7], as well as state-of-the-art DNN compilers such as Ladder [51] and PyTorch-Inductor (v2.4.0, with Triton v3.0.0) [10, 49]. PyTorch integrates bitsandbytes [30], an official backend in HuggingFace transformers, for $W_{FP4}A_{FP16}$ precision. Additionally, we evaluate PipeThreader against TensorRT (v10.0.1) [6], a vendor-specific inference library for NVIDIA GPUs. We also compare PipeThreader with libraries cuBLAS [2] and rocBLAS (on ROCm GPUs) [1] for MatMul, Ladder and bitsandbytes library for low-precision MatMul operations, MIOpen [36] for Conv2D, and expert-optimized FlashAttention-3 [46] kernel written using CUTLASS [5] templates for attention operation. For LLM



Figure 12: Operator performance on NVIDIA H100 GPU.

and Mamba models, we further benchmark PipeThreader against vLLM (v0.6.3) [37], the most popular library for LLM inference. Average performance metrics, such as speedup, are calculated using the geometric mean across all experiments. All evaluations begin with warm-up iterations, followed by repeated execution of each workload for a minimum of 5 seconds to ensure accurate and stable results.

6.2 Operator Performance on NVIDIA H100

Figure 12 shows the performance of all operator configurations in the microbenchmark. The x-axis represents different operators, and the y-axis indicates the normalized latency relative to PipeThreader.

MatMul. The first row of Figure 12 shows the performance of PipeThreader and other baselines on MatMul operators derived from LLAMA3-8B (M0-M7) and LLAMA3-70B (M8-M15). Although the existing compiler or libraries already provide well-optimized MatMul kernels, the result shows that PipeThreader still obtains a significant speedup. Compared to PyTorch, Triton, and Ladder, PipeThreader achieves average speedups of 1.24× (up to 1.40×), 1.13× (up to 1.26×), and 2.07× (up to 2.25×), respectively. This improvement stems from PipeThreader's ability to leverage its sTask abstraction to model MatMul as a pipeline of `load` and `mma` sTasks and fully explore advanced scheduling opportunities. Notably, PipeThreader matches cuBLAS MatMul performance, with an average speedup of 1.06×. It also achieves over 750 TFLOPS on most of these MatMul operators, approaching the theoretical TensorCore peak performance of the H100 GPU.

Convolution. PipeThreader implements convolutions using implicit GEMM [39], which also leverages the pipeline optimization of the `load` and `mma`. As demonstrated in the second row of Figure 12, PipeThreader significantly outperforms the baselines on the convolution operators derived from the ResNet-50 model (with batch sizes of 1 and 128). PipeThreader achieves an average performance improvement of 1.94× (up to 3.52×) over PyTorch and 2.56× (up to 8.66×) over Ladder, respectively. Since the official Conv2D implementation is not provided, we implement a Conv2D kernel using Triton and perform auto-tuning to achieve its best performance. Compared to Triton, PipeThreader delivers an average speedup of 1.85× (up to 2.47×).

Low-bit MatMul. The third row of Figure 12 shows the performance of low-bit MatMul operators derived from $W_{\text{FP4}}A_{\text{FP16}}$-quantized LLAMA3-8B (DM0-DM7) and LLAMA3-70B (DM8-DM15). As current TensorCores do not directly support $W_{\text{FP4}}A_{\text{FP16}}$ MatMul, the low-bit MatMul must first cast data to FP16 on CUDA cores, which introduces

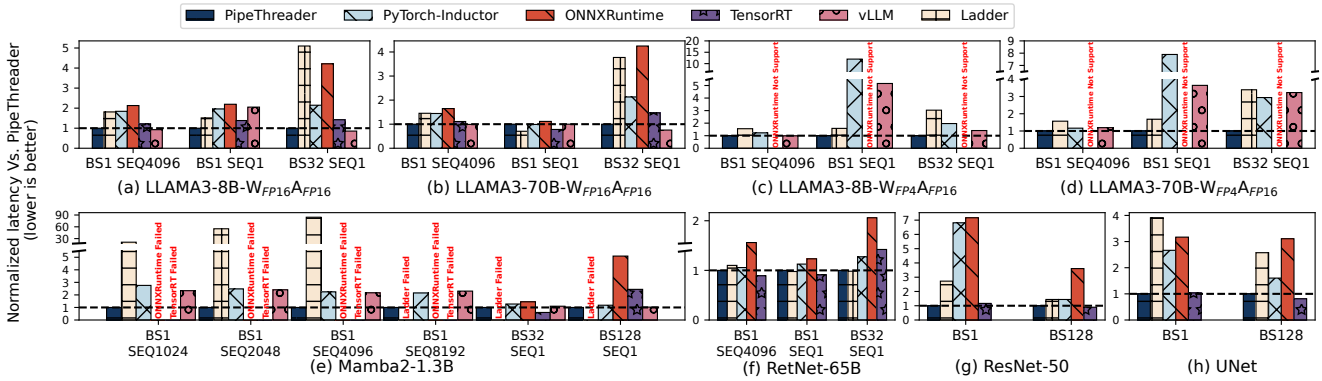


Figure 13: End-to-end performance on the NVIDIA H100 GPU.

Configuration	Pipe-Threader	PyTorch-Inductor	ONNX-Runtime	TensorRT	vLLM	Ladder
BS=1, SEQ=4k	3.29	6.06	7.01	3.99	3.05	5.96
BS=1, SEQ=1	0.29	0.56	0.63	0.40	0.59	0.43
BS=32, SEQ=1	1.29	2.78	5.47	1.84	1.11	6.62

Table 2: Latency (in milliseconds) of LLAMA3-8B-$W_{\text{FP16}}A_{\text{FP16}}$ on the NVIDIA H100 GPU.

an additional `dequant` stage. With more pipeline stages, we observe that PipeThreader gains larger speedups than the ones achieved on standard MatMul. For example, PipeThreader outperforms PyTorch (with bitsandbytes) and Ladder by 3.92× (up to 4.76×) and 2.48× (up to 3.81×) on low-bit MatMul, while the average speedups on standard MatMul are only 1.24× and 2.07×, respectively.

FlashAttention and FlashDecoding. Compared to MatMul and low-bit MatMul, FlashAttention involves even deeper computation stages, opening up a larger optimization space for PipeThreader. PipeThreader can automatically find effective scheduling schemes by exploring the new optimization space defined by an sTask-graph and the new exposed hardware capabilities. The fourth row of Figure 12 shows the performance of FlashAttention operators derived from LLAMA3-8B (FA0-FA9) and LLAMA3-70B (FA10-FA19). The evaluations cover sequence lengths ranging from 512 to 8k, with batch sizes of 1 and 64, including configurations with and without causal masking. PipeThreader demonstrates an average performance improvement of 1.36× (up to 1.50×) over Triton, compared to 1.13× (up to 1.26×) for MatMul operation. This highlights the system's ability to exploit the larger optimization space inherent in FlashAttention's more complex computation pipeline.

Such a general scheduling capability enables PipeThreader to achieve comparable performance to model-specific implementations optimized by experts, and in some configurations, even outperform them. FlashAttention-3, a manually optimized attention kernel tailored for NVIDIA Hopper GPUs, is an example of such an expert-designed approach. PipeThreader achieves an average performance improvement of 1.07× (up to 2.18×) over FlashAttention-3. As a hand-crafted approach, FlashAttention-3 cannot efficiently optimize for varying workloads. We observe that, particularly for smaller sequence lengths, FlashAttention-3 performs suboptimally due to fixed tile sizes. PyTorch uses the hand-crafted FlashAttention-2 kernel without incorporating finer-grained pipelining. Compared to PyTorch, PipeThreader achieves an average speedup of 1.82× (up to 2.29×).

FlashDecoding operators are selected from the LLAMA3-8B (FD0, FD1) and LLAMA3-70B (FD2, FD3) models with batch size set to 1 and context length set to 8192 to simulate decoding scenarios. Compared to FlashAttention-3 and Triton, PipeThreader achieves an average speedup of 1.12× (up to 1.23×) and 2.27× (up to 3.06×), respectively.

Linear Attention. The fifth and sixth rows of Figure 12 present the key linear attention operations of the Mamba2 model: ChunkScan and ChunkState operations, respectively. We compare PipeThreader with the official Triton implementation. The configurations we test have sequence lengths ranging from 1k to 16k, with batch sizes of 1 or 64. PipeThreader achieves average speedups of 1.71× (up to 1.99×) and 1.98× (up to 2.59×) over Triton for ChunkScan and ChunkState operations, respectively. Also, Triton fails on some configurations, such as sequence lengths of 8k (CC14, CT14) and 16k (CC7, CC15, CT7, CT15). These results emphasize PipeThreader's adaptability in handling emerging DNN operations, eliminating the need for hand-crafted implementations.

6.3 End-to-end Performance on NVIDIA H100

Figure 13 shows the end-to-end performance of all eight DNN models on the NVIDIA H100 GPU. Due to GPU memory constraints, we evaluate the inference latency of large language models like LLAMA3, Mamba2, and RetNet using a single decoder layer, which serves as a proxy for full-model performance, as all layers are identical and latency scales linearly with the number of layers.

LLM models. On the FP16-precision LLAMA3-8B and LLAMA3-70B models, PipeThreader achieves average

Pattern	Configuration	Triton	PT-Decouple	PT-Joint
ChunkScan	BS=1, SEQ=8k	0.158	0.198	0.108
ChunkScan	BS=64, SEQ=8k	13.332	12.150	6.981
FlashAttention	BS=1, SEQ=4k	0.639	0.851	0.438
FlashAttention	BS=64, SEQ=4k	41.681	48.762	30.416

Table 3: Latency (in milliseconds) comparison for decouple and joint sTask-graph optimization.

Pattern	Configuration	PipeThreader	Triton	CUTLASS
MatMul	M,N,K=8k,8k,8k	0.13	0.17	3.36
FlashAttention	BS=64, SEQ=8k	5.26	0.74	/
ChunkScan	BS=1, SEQ=16k	3.92	1.28	/
ChunkState	BS=1, SEQ=16k	3.71	1.05	/

Table 4: Compilation time (in minutes) on H100.

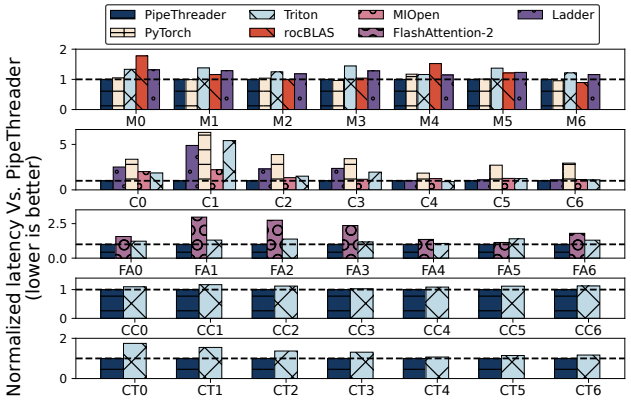


Figure 14: Operator performance on AMD MI300X GPU.

speedups of 2.17× and 2.45× over Ladder and ONNXRuntime, respectively. Ladder's schedule policy is unable to effectively represent and generate the FlashAttention kernel, while ONNXRuntime does not natively provide support for FlashAttention, leading to suboptimal performance. Although PyTorch-Inductor, TensorRT, and vLLM integrate industrially popular FlashAttention kernel as their backend, PipeThreader on average outperforms them by 1.79× (up to 2.15×), 1.28× (up to 1.47×), and 1.10× (up to 2.05×), respectively. This improvement is attributed to PipeThreader's ability to explore more efficient pipeline configurations tailored to operations in the model (e.g., MatMul and FlashAttention). In $W_{\text{FP4}}A_{\text{FP16}}$ quantization scenarios, PipeThreader outperforms Ladder, PyTorch-Inductor, and vLLM with an average speedup of 2.01× (up to 3.39×), 3.03× (up to 11.98×), and 2.16× (up to 5.16×), respectively, as it optimizes the pipeline scheduling for both low-bit MatMul and FlashAttention. We also include a subset of absolute performance in Table 2.

Linear Attention models. On the Mamba2-1.3B model, Ladder, ONNXRuntime, and TensorRT fail to generate efficient linear attention kernels, leading to memory errors under certain configurations. PyTorch-Inductor, which uses Triton as the backend, is capable of running fused linear attention but delivers suboptimal performance compared to PipeThreader. PipeThreader outperforms PyTorch-Inductor, ONNXRuntime, TensorRT, vLLM, and Ladder by 1.92× (up to 2.76×), 2.71× (up to 5.10×), 1.21× (up to 2.44×), 1.78× (up to 2.41×), and 45.93× (up to 84.41×), respectively, given its ability to explore more efficient pipeline configurations tailored to operations in the model.

For the linear attention model RetNet-65B, compared to PyTorch-Inductor, ONNXRuntime, TensorRT, and Ladder, PipeThreader achieves speedups by 1.16× (up to 1.31×), 1.60× (up to 2.12×), 1.06× (up to 1.46×), and 1.04× (up to 1.10×), respectively. On the RetNet-65B model, the speedup achieved by PipeThreader is relatively modest. This is because the RetNet-65B model's large attention head dimensions (256 for query and key, 432 for value) lead to high shared memory usage, limiting pipeline scheduling optimization.

CNN models. For ResNet-50 and UNet, by generating more efficient Conv2D kernels, PipeThreader achieves speedups of 2.01×, 2.54×, and 3.99× in end-to-end inference latency over Ladder, PyTorch-Inductor, and ONNXRuntime, respectively. Compared to TensorRT, PipeThreader achieves comparable performance (0.97×).

6.4 Evaluation on Scheduling Policy

Joint optimization. Our scheduling policy jointly optimizes partitioning and pipeline scheduling of the sTask-graph. To demonstrate the benefits, we create a variant of PipeThreader: "PT-decouple", which optimizes the partitioning (e.g., for high memory utilization in a single sTask) and pipeline scheduling (e.g., for high overlapping) in independent optimization passes. As shown in Table 3, on the Mamba2-ChunkScan (BS=64, SEQ=8k) operator, with the PT-decouple, the compiler focuses on maximizing data reuse and selects a larger tile shape (e.g., 64×128). However, this larger tile shape limits the sTask-graph's ability to achieve effective pipeline parallelism on one EU, resulting in an execution time of 12.150 ms. By jointly optimizing both the sTask-graph partitioning and scheduling, the compiler selects a smaller tile shape (e.g., 64×64), which facilitates more efficient pipelining and reduces the execution time to 6.981 ms.

Compilation time. The joint optimization would require a relatively longer compilation time. Table 4 presents the compilation times of PipeThreader for several typical configurations in FlashAttention and Mamba2. Since all task partitions are generated in line 2 of Figure 9, the scheduling process can be parallelized to accelerate compilation. For simple kernels with few pipeline depths, such as MatMul, PipeThreader compilation takes only 0.13 minutes, compared to 0.17 minutes and 3.36 minutes for Triton and CUTLASS, respectively. For complex fused kernels such as FlashAttention, PipeThreader still achieves a short compilation time of 5.26 minutes, even though it explores a large pipeline search space over Triton.

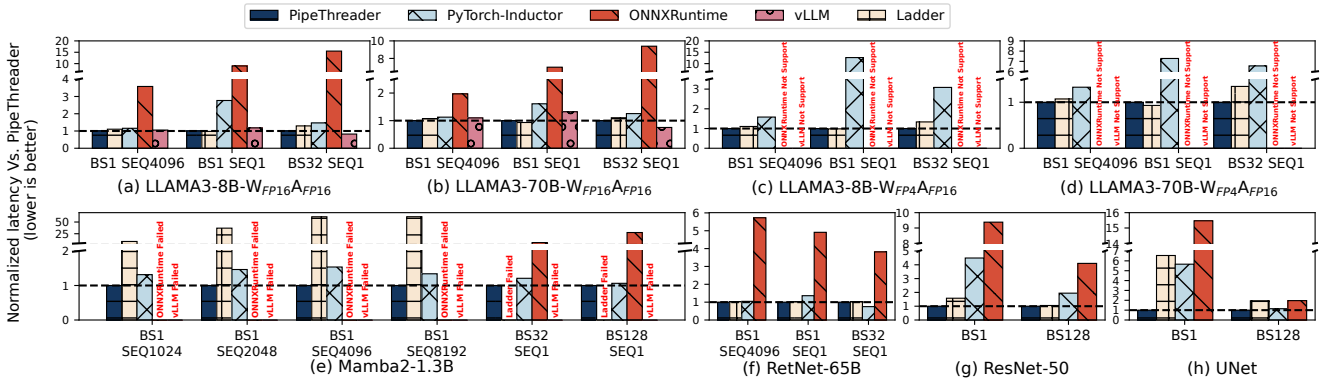


Figure 15: End-to-end performance on AMD Instinct MI300X GPU.

6.5 Evaluation on AMD ROCm GPUs

Operator performance. We benchmark the AMD MI300X GPU using a subset of operators selected from the microbenchmark suite originally designed for the NVIDIA H100 GPU. The evaluation focuses on key operators: MatMul (compared against PyTorch, rocBLAS, Triton, and Ladder), Conv2D (compared against PyTorch, MIOpen, Triton, and Ladder), FlashAttention (compared against FlashAttention-2 and Triton), and Linear Attention (compared against Triton).

Figure 14 shows that PipeThreader achieves speedups of 1.16× to 5.42× over Triton, and up to 6.21× over PyTorch, across different types of operators. PipeThreader surpasses rocBLAS and MIOpen on MatMul and Conv2D, with gains of up to 1.77× and 2.21×, respectively. PipeThreader also achieves up to 2.82× speedup over FlashAttention-2. Furthermore, PipeThreader outperforms Ladder with an average speedup of 1.45×, demonstrating its efficiency and scalability.

End-to-end performance. We evaluate PipeThreader on the AMD Instinct MI300X GPU against Ladder, PyTorch-Inductor, ONNXRuntime, and vLLM. Figure 15 shows the end-to-end performance results across eight models, including LLAMA3-8B and LLAMA3-70B in both $W_{\text{FP16}}A_{\text{FP16}}$ and $W_{\text{FP4}}A_{\text{FP16}}$ formats, as well as Mamba2-1.3B, RetNet-65B, ResNet-50, and UNet.

On the FP16-precision LLAMA3-8B and LLAMA3-70B models, PipeThreader achieves speedups of 1.48× (up to 2.77×), 6.33× (up to 15.51×), 1.02× (up to 1.32×), and 1.07× (up to 1.29×) over PyTorch-Inductor, ONNXRuntime, vLLM, and Ladder, respectively. On the $W_{\text{FP4}}A_{\text{FP16}}$ LLAMA3-8B and LLAMA3-70B models, PipeThreader achieves speedups of 3.97× (up to 12.66×) and 1.12× (up to 1.34×) over PyTorch-Inductor and Ladder, respectively, while ONNXRuntime and vLLM do not support $W_{\text{FP4}}A_{\text{FP16}}$ quantization on the ROCm platform.

For Mamba2-1.3B, PipeThreader achieves an impressive average speedup of 32.93× (up to 61.33×) compared to Ladder. This significant performance gain is primarily attributed to Ladder's inability to fuse the Linear Attention component. In comparison to PyTorch-Inductor, PipeThreader delivers an average speedup of 1.31× (up to 1.54×). On the RetNet-65B model, PipeThreader achieves speedups of 1.03× (up to 1.36×), 4.75× (up to 5.73×), and 1.01× (up to 1.02×) over PyTorch-Inductor, ONNXRuntime, and Ladder, respectively.

On traditional CNN models (ResNet-50 and UNet), PipeThreader achieves speedups of 2.74× (up to 5.66×), 5.84× (up to 15.47×), and 2.14× (up to 6.54×) compared to PyTorch-Inductor, ONNXRuntime, and Ladder, respectively.

MI300X GPU has relatively lower asynchronous capabilities and smaller shared memory than the NVIDIA H100 GPU, which reduces the potential performance (e.g., speedups over baselines) for pipeline parallelism enabled by PipeThreader.

7 Discussion

Advantages over hand-crafted kernels. PipeThreader demonstrates fundamental advantages over hand-crafted implementations, especially in automated pipeline scheduling and cross-architecture portability.

First, it eliminates the need for expert-level manual tuning. Designing efficient pipeline schedules by hand is error-prone, time-consuming, and sensitive to input configurations. Even expert-crafted kernels like FlashAttention-3 (FA3) initially lacked support for certain dimensions (e.g., head size 256), illustrating the difficulty. PipeThreader automates this process, systematically exploring the scheduling space under hardware constraints. It achieves up to 2.18× speedup over FA3 and outperforms vLLM's Triton-based Mamba2 by 2.41×.

Second, it generalizes well across hardware. While hand-tuned kernels are often tightly coupled to specific platforms, especially NVIDIA GPUs, PipeThreader can achieve significant gains on AMD hardware. Its abstraction also maps naturally to TPU-like architectures [34, 35] (e.g., TPU cores and DMA engines), enabling efficient pipelined execution.

Finally, it lowers the barrier to high performance. On Multi-head Latent Attention (MLA) [40, 41], PipeThreader delivers up to 5× speedup over Triton with only 80 lines of Python, matching DeepSeek's 500+ line CUDA implementation [4], but with far less development effort.

Scale to multi-GPU. PipeThreader can naturally scale to multi-GPU and interplay with collective communications introduced by tensor parallelism, through including the 1) communication units (e.g., RDMA, NVLINK, IB) between GPUs as sEUs, and 2) collective communications as sTasks. In this way, PipeThreader can reuse the policy to search for the efficient pipelining between collective communication and computation at the GPU kernel level, and scale to multi-GPU or multi-node environments. Current results show that PipeThreader performs on par with state-of-the-art systems such as TileLink [55] on common communication patterns.

Supporting new devices. We find that widely used hardware (e.g., NVIDIA/AMD GPUs or TPUs) aligns with the sEU abstractions, which contain even sets of all the sEUs. To compile sTask-graphs onto the device, the programming model only requires implementing the `Execute` interface of each sEU with their own load/store/compute instructions (Figure 4). The device virtualization resembles the hardware abstraction of Roller [56] and Welder [47], but further exposes the fine-grained heterogeneous sEUs.

MoE FFN kernels. PipeThreader can also support grouped MatMul in MoE FFN kernels. Unlike batched MatMul, each group can have a different shape. To handle this, PipeThreader can decompose each group into an independent sTask-subgraph with its own input shapes and apply policies separately, rather than sharing a single schedule.

8 Related Work

Deep learning compiler and frameworks. Most existing DNN compilers abstract hardware as homogeneous execution units (EUs). Rammer [43] introduces the concept of rTasks, designed for parallel execution across EUs, while Welder [47] focuses on holistic memory optimization through vertical fusion. In contrast, PipeThreader introduces sTasks and sEUs, explicitly exposing the heterogeneity of hardware to enable the optimization and scheduling of pipeline parallelism.

Operator fusion has been widely adopted in DNN compilers such as TVM [19], Ansor [54], XLA [13], and TensorRT [6] to reduce memory overhead, resulting in deeper computational stages. Compilers like Triton [49], Welder [47], Roller [56], Cocktailer [52], TensorIR [11], ThunderKittens [12], FractalTensor [42], and Ladder [51] base their scheduling optimizations on tile abstractions. However, these primarily focus on spatial tiling to enhance data locality, enabling data parallelism across EUs but neglecting opportunities for pipeline parallelism across sEUs.

While efforts like Triton [49] and CUTLASS [5] incorporate pipelined execution, they rely on ad hoc rules for specific operators and fail to generalize across diverse workloads. PipeThreader addresses these limitations by introducing abstractions that enable automatic scheduling and optimization of pipeline parallelism.

Frameworks like ALCOP [33] focus on pipelining between data loading and computation to optimize memory hierarchy utilization. However, they fail to fully exploit the heterogeneity of modern computation units or to explore pipeline scheduling for workloads with deep computational stages, such as FlashAttention. PipeThreader addresses this gap by introducing finer-grained abstractions that enable comprehensive pipeline optimization across heterogeneous hardware components.

Optimizations for specific patterns. Due to the lack of sTask and sEU abstractions in existing compilers, pipeline parallelism optimization is often manually tailored to specific patterns. For instance, FlashAttention [46] and Hopper MatMul in CUTLASS [5] provide pattern-specific schedules but require significant manual effort. Additionally, FlashAttention provides separate schedules for different inputs, while CUTLASS [5] requires users to profile and select the best schedule. In comparison, PipeThreader generalizes pipeline parallelism by leveraging sTask and sEU abstractions, enabling automatic scheduling across a wide range of operators and configurations without requiring manual intervention.

Distributed deep learning frameworks. Centauri [18], PrimePar [50], and TileLink [55] improve communication-computation overlap through hierarchical scheduling, temporal tensor partitioning, and tile-based abstractions, respectively. PipeThreader can model communication and computation as separate sTasks, allowing it to represent the scheduling strategies proposed in these works, while enabling broader scheduling optimization.

9 Conclusion

As DNN models grow larger and specialized heterogeneous hardware units emerge, hardware schedulers are no longer sufficient for efficient pipeline execution. This paper introduces PipeThreader, a DNN compiler that enables software-defined pipelining via an sTask-graph abstraction and hierarchical hardware capabilities, combining virtualized EUs and specialized sEUs. With key scheduling primitives, PipeThreader automates pipeline scheduling, achieving comparable or superior performance to state-of-the-art methods like FlashAttention on H100 and AMD GPUs, while generalizing to emerging models such as Mamba2. We believe PipeThreader lays the groundwork for further advancements in compiler-based optimizations, paving the way for efficient utilization of evolving GPU architectures and DNN workloads.

Acknowledgement

We thank anonymous reviewers and our shepherd, Deepti Raghavan, for their extensive suggestions. This work was partially supported by National Natural Science Foundation of China under Grant No. 92464301. Zhi Yang is the corresponding author.

References

[1] A BLAS implementation on top of ROCm. https://rocmdocs.amd.com/en/latest/ROCm_Tools/rocblas.html.

[2] CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/index.html.

[3] CUDA Driver API. http://docs.nvidia.com/cuda/cuda-driver-api.

[4] FlashMLA. https://github.com/deepseek-ai/FlashMLA.

[5] NVIDIA cutlass. https://github.com/NVIDIA/cutlass.

[6] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[7] ONNX Runtime. https://github.com/microsoft/onnxruntime.

[8] Overlapping CUDA Cores and Tensor Cores. https://forums.developer.nvidia.com/t/overlapping-cuda-cores-and-tensor-cores/288774/2.

[9] PTX ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/.

[10] PyTorch. https://pytorch.org/.

[11] TensorIR. https://discuss.tvm.apache.org/t/rfc-tensorir-a-schedulable-ir-for-tvm/7872.

[12] ThunderKittens. https://github.com/HazyResearch/ThunderKittens.

[13] XLA. https://www.tensorflow.org/xla.

[14] Inc. Advanced Micro Devices. Introducing amd cdna™ architecture. Technical report, Advanced Micro Devices, Inc., 2020.

[15] Inc. Advanced Micro Devices. Introducing amd cdna™ 2 architecture. Technical report, Advanced Micro Devices, Inc., 2021.

[16] Inc. Advanced Micro Devices. Amd cdna™ 3 architecture. Technical report, Advanced Micro Devices, Inc., 2023.

[17] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: Leveraging warp specialization for high performance on gpus. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 119–130, 2014.

[18] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 178–191, 2024.

[19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[20] NVIDIA Corporation. Nvidia tesla v100 gpu architecture. Technical report, NVIDIA Corporation, 2017.

[21] NVIDIA Corporation. Nvidia a100 tensor core gpu architecture. Technical report, NVIDIA Corporation, 2020.

[22] NVIDIA Corporation. Nvidia h100 tensor core gpu architecture. Technical report, NVIDIA Corporation, 2023.

[23] NVIDIA Corporation. Cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2024.

[24] NVIDIA Corporation. Nvidia blackwell architecture. Technical report, NVIDIA Corporation, 2025.

[25] Neal C Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W Keckler. Wasp: Exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–16. IEEE, 2024.

[26] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[27] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[28] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

[29] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[30] Bitsandbytes Foundation. bitsandbytes, 2024.

[31] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[33] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems*, 5:680–694, 2023.

[34] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[35] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. Miopen: An open source library for deep learning primitives, 2019.

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

[38] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[39] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International conference on parallel processing (ICPP)*, pages 67–76. IEEE, 2016.

[40] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[41] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[42] Siran Liu, Chengxiang Qi, Ying Cao, Chao Yang, Weifang Hu, Xuanhua Shi, Fan Yang, and Mao Yang. Uncovering nested data parallelism and data reuse in dnn computation with fractaltensor. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 160–177, 2024.

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

[44] OpenAI. Introducing chatgpt, 2022. Available: https://openai.com/blog/chatgpt.

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[46] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[47] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[48] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

[49] Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*, page 10–19. Association for Computing Machinery, New York, NY, USA, 2019.

[50] Haoran Wang, Lei Wang, Haobo Xu, Ying Wang, Yuming Li, and Yinhe Han. Primepar: Efficient spatial-temporal tensor partitioning for large transformer model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 801–817, 2024.

[51] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, et al. Ladder: Enabling efficient {Low-Precision} deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 307–323, 2024.

[52] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 681–699, Boston, MA, July 2023. USENIX Association.

[53] Jie Zhao and Albert Cohen. Flextended tiles: A flexible extension of overlapped tiles for polyhedral compilation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–25, 2019.

[54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[55] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025.

[56] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.

WLB-LLM: Workload-Balanced 4D Parallelism for Large Language Model Training

Zheng Wang[1,2], Anna Cai[2], Xinfeng Xie[2], Zaifeng Pan[1], Yue Guan[1], Weiwei Chu[2], Jie Wang[2], Shikai Li[2], Jianyu Huang[2], Chris Cai[2], Yuchen Hao[2], Yufei Ding[1,2]
University of California, San Diego[1]
Meta[2]

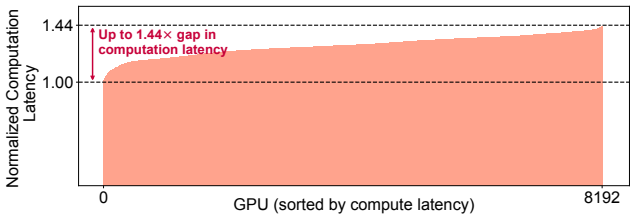
Abstract

In this work, we present *WLB-LLM*, a *W*ork*L*oad-*B*alanced 4D Parallelism for *L*arge *L*anguage *M*odel Training. We first thoroughly analyze the workload imbalance issue in LLM training and identify two primary sources of imbalance at the pipeline parallelism and context parallelism levels. Then, to address the imbalance issue, at the pipeline parallelism level, *WLB-LLM* incorporates a workload-aware variable-length document packing method to balance the computation and communication workload across micro-batches. Additionally, at the context parallelism level, *WLB-LLM* introduces a novel fine-grained per-document sharding strategy, ensuring each worker within a context parallelism group has an identical workload. Comprehensive experiments under different model scales demonstrate that *WLB-LLM* significantly mitigates the workload imbalance during 4D parallelism LLM training and achieves an average speedup of 1.23× when applying *WLB-LLM* in our internal LLM training framework.

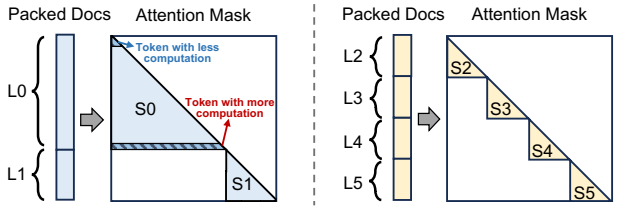
1 Introduction

Large language models (LLMs) have been widely adopted as the backbone of various applications, such as coding assistants [27, 48], language translation [52], and chatbots [3, 31]. The remarkable capabilities and promising potential of LLMs have sparked a competition among big tech companies to train LLMs with higher intelligence and broader versatility [2, 6, 41]. As the scale of LLMs and length of context window continue to grow larger [14], the training of LLMs consumes a significant amount of computing power [33]. For instance, Meta reports that the training of the LLaMA3-405B model uses 16K H100 GPUs for several months [6]. The tremendous computational cost of LLM training makes every improvement in end-to-end training efficiency translate to substantial savings.

Unfortunately, a significant portion of GPUs are underutilized during large-scale LLM training jobs due to the workload imbalance issue. To illustrate this, Figure 1 (a) shows the normalized computation latency on each GPU during a 405B LLM training job executed across 8K H100 GPUs with a context window of 128K. It can be observed that the computation latency exhibits significant variance among GPUs. The slowest GPU suffers a computation latency that is 1.44× longer than the others. This disparity in computing latency causes a substantial degradation in training efficiency as the synchronized nature of training requires all other GPUs to wait for the slowest GPU to finish.



(a) Normalized computation latency in an 8K-GPU 405B LLM training job.



(b) Reason of imbalance: input-dependent nature of attention computation and the varying input document length.

Figure 1: Observed workload imbalance issue in large-scale LLM training jobs and the reason of workload imbalance.

The root cause of the workload imbalance lies in the input-dependent nature of attention computation and the varying input document length of the training data. As shown in Figure 1 (b), input sequences are composed of multiple input documents. To ensure model quality in long-context training, attention masks are applied to prevent attention computation between tokens from different documents within the same

sequence [6]. This approach introduces heterogeneity in per-token arithmetic intensity, as tokens at the tail positions of long documents require more attention computation. As a result, input sequences containing long documents incur significantly higher computation workloads, even with the same total sequence length.

However, existing LLM training frameworks [23, 26, 35, 38, 39] fail to recognize the heterogeneity in per-token arithmetic intensity. Specifically, state-of-the-art LLM training solutions employ a 4D parallelism paradigm that combines data parallelism (DP) [35], pipeline parallelism (PP) [25], context parallelism (CP) [30], and tensor parallelism (TP) [38]. The input documents are packed into sequences with fixed length at the DP and PP levels, and then sharded and distributed into chunks at the CP and TP levels. This fixed and static training flow treats all input tokens homogeneously and assigns each GPU an equal number of tokens, inevitably resulting in workload imbalance between GPUs. Furthermore, the trend of larger models and longer context windows exacerbates this issue, increasing the likelihood of an extremely long document appearing in the input batches, thereby delaying the entire training step.

An intuitive approach to address the workload imbalance issue is to shuffle and repack input documents to distribute the computation workload more evenly across micro-batches. However, this method is not effective and practical for two main reasons: *Firstly*, achieving effective balance through shuffling and repacking requires a sufficiently large packing window spanning multiple global batches. This impacts the randomness of data sampling and loading, which can potentially affect model convergence during training (as discussed in Section 3.3). *Secondly*, shuffling and repacking only address workload imbalances across micro-batches and cannot resolve intra-document imbalances caused by sequence sharding. In 4D parallelism training, input sequences are divided into chunks and distributed across different GPUs. Document chunks that include the tail end of a document incur a higher computational workload because their tokens must attend to more preceding tokens, causing an intra-document workload imbalance across GPUs.

To overcome the challenges mentioned above and address the severe workload imbalance issue in large-scale LLM training, we propose a flexible and input-aware document packing and sharding approach for the 4D parallelism training paradigm. The data packing and sharding will no longer output micro-batches with a fixed number of tokens. Instead, each GPU aims to get input tokens that have an equal amount of total computation and communication workload. Additionally, to minimize the impact on data randomness, we propose to only adjust the execution order of extremely long documents. This is based on our observation that the tokens of long documents account for only a small proportion of the total training tokens but have the most significant impact on workload imbalance.

Based on the above design insight, we build *WLB-LLM*, a *W*ork*L*oad-*B*alanced 4D Parallelism for *L*arge *L*anguage *M*odel Training. We begin by thoroughly analyzing workload imbalance in LLM training under 4D parallelism (§3), identifying two primary sources of imbalance across specific parallelism hierarchies: (1) Imbalance across micro-batches at the pipeline parallelism level and (2) Imbalance across document shards at the context parallelism level. To address these imbalances, *WLB-LLM* provides novel solutions tailored specifically to each parallelism level. **At the PP level**, *WLB-LLM* introduces variable-length document packing, allowing shorter documents to be combined to form longer sequences, thereby aligning the total computation workload to that of a single long document (§4). Additionally, *WLB-LLM* adaptively delays the execution of extremely long documents, achieving near-optimal workload balance across micro-batches while maintaining a relatively low per-token delay, thereby preserving the randomness of the data loader. **At the CP level**, *WLB-LLM* incorporates a novel per-document sharding strategy, ensuring each worker within a CP group has an equal workload (§5). Furthermore, we observe a tradeoff between kernel efficiency and sharding balance with per-document sharding. To maximize overall performance, we propose a heuristic algorithm that adaptively selects the most efficient sharding strategy based on the input sequence at runtime.

In summary, this paper makes the following contributions:

- To the best of our knowledge, we are the first to identify, analyze, and address workload imbalance issues in large-scale LLM training with 4D parallelism.
- At the PP level, we design a variable-length input packing and adaptive outlier delay strategy to achieve near-optimal workload balance across micro-batches, while minimizing impacts on data loading randomness and model convergence.
- At the CP level, we implement a fine-grained per-document sharding method with an adaptive sharding selection mechanism to achieve the most efficient sharding for each input micro-batch at runtime.
- We conduct comprehensive evaluations and demonstrate that *WLB-LLM* achieves an average speedup of 1.23× across various model scales and context window sizes.

2 Background

In this section, we provide the background of 4D parallelism LLM training and discuss the characteristics of input documents in long-context LLM training.

2.1 4D Parallelism LLM Training

Training extremely large LLMs with billions or even trillions of parameters is challenging, involving a significant amount

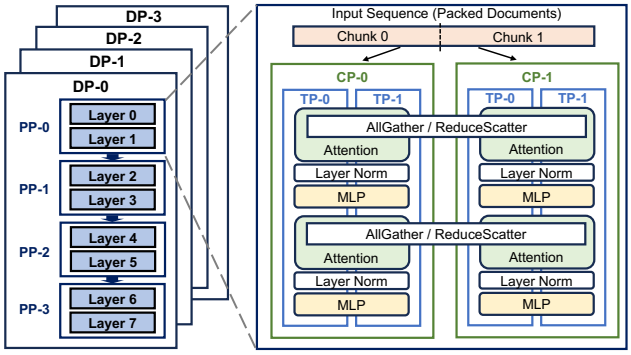


Figure 2: Overview of 4D parallelism for LLM training.

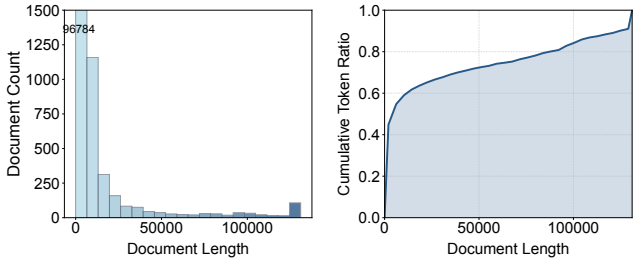


Figure 3: Characterization of input documents: Distribution of input document lengths (left) and cumulative token ratio by document length (right).

of engineering effort to tune the multi-level parallelism [55]. The state-of-the-art LLM training framework features a 4-dimensional parallelism [9, 12], including data parallelism (DP) [35], pipeline parallelism (PP) [25], context parallelism (CP) [30], and tensor parallelism (TP) [38]. Figure 2 presents an example of a (TP=2, CP=2, PP=4, DP=4) 4D parallelism.

Data Parallelism: In DP, the input global batch is partitioned and distributed across DP workers, with each worker owning a part of the global batch. By default, model parameters are duplicated across DP workers. Some advanced techniques like ZeRO [35] and FSDP [32] partition model parameters, gradients, and optimizer states across DP workers to reduce memory consumption. During training, each DP worker computes parameter updates using its local batch independently and then synchronizes gradients with other workers via AllReduce (or ReduceScatter when using FSDP).

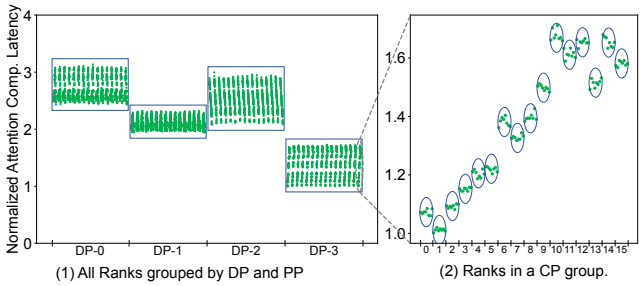
Pipeline Parallelism: Within each DP worker, the devices are further partitioned into multiple PP workers through pipeline parallelism. In PP, the model is split in a layer-wise manner, with each PP worker owning several chunks of layers. The input batches of a DP worker are also divided into multiple micro-batches. During training, a micro-batch traverses through all PP workers from first to last, and then reverses direction in the backward pass. Peer-to-peer (P2P) communication is required to send activations and gradients during forward and backward passes, respectively.

Context Parallelism: CP is designed to address the large memory consumption of activations in long-context training. As shown in Figure 2, CP duplicates the model parameters but shards the input and activations along the sequence length dimension across CP workers. CP operates either in a ring-based manner using P2P communication [24] or involves AllGather communication during the forward phase to collect the KV (key and value) tensors of all tokens, and ReduceScatter communication for the gradients of the KV tensors in backward [6]. All other operators, such as Linear and LayerNorm, operate independently on all workers, similar to DP.

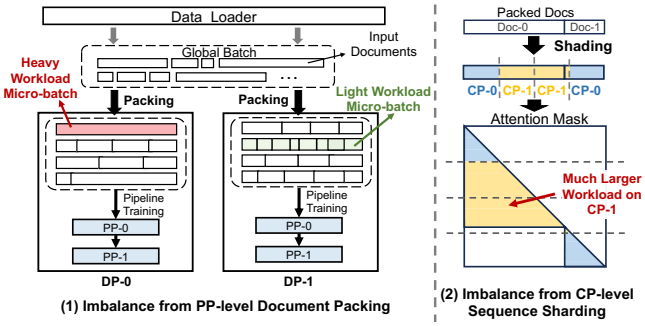
Tensor Parallelism: TP splits the model parameters within a layer (e.g., attention, FFN) and distributes them across multiple TP workers. TP is often applied in conjunction with Sequence Parallelism (SP) [15] to further split the input tensor and activations. In this paper, when referring to TP, we mean both TP and SP by default. With TP, each GPU only has part of the input and parameters, resulting in intensive AllGather and ReduceScatter communication during training. Therefore, TP is typically applied within a single node, while other levels of parallelism are applied across nodes.

2.2 Varying Input Document Length

In large-scale LLM training, the lengths of input documents could vary significantly, especially when using a large context window. To illustrate this, we profile the characteristics of training data in our 128K context length training job, as shown in Figure 3. **From a per-document perspective**, input document lengths distribution is highly skewed. As shown in the left part of Figure 3, the majority of input documents are relatively short, while some extremely long documents exist, with the longest reaching the full context window size. The presence of an extremely long document in an input batch can easily lead to significant workload imbalances across micro-batches. This observation highlights the need for an input-aware solution that dynamically balances workloads by accounting for variations in document length. **From a per-token perspective**, we compute the cumulative token ratio across different document lengths. As shown in the right part of Figure 3, the majority of training tokens come from relatively short documents. For instance, documents shorter than half the context window contribute over 75% of the total training tokens. Although long documents significantly impact workload balance, the tokens from these long documents constitute only a small proportion of the training dataset. This provides an opportunity to adaptively delay the execution of extremely long documents to mitigate workload imbalance, while minimizing the impact on data sampling randomness for the majority of tokens.



(a) Imbalance Analysis (TP=8, CP=16, PP=16, DP=4): (1) Normalized computation latency (group by DP and PP); (2) Normalized computation latency in a CP group.



(b) Document packing at PP level and sequence sharding at CP level.

Figure 4: The workload imbalance comes from the PP-level document packing and CP-level sequence sharding.

3 Motivation

To motivate the design of *WLB-LLM*, we first conduct an in-depth analysis of workload imbalances across different parallelism hierarchies in 4D parallelism LLM training. We then introduce a baseline solution that shuffles and repacks input documents across batches. Lastly, we discuss why this baseline solution cannot fully eliminate workload imbalances and investigate the tradeoff between input packing balance and model convergence.

3.1 Imbalance Analysis

After analyzing the performance trace from our internal 8K-GPU, 128K context window training job for a 405B LLM, we identify two primary causes of workload imbalance: (1) workload imbalance among micro-batches at the PP level, and (2) imbalance across sequence shards at the CP level. To demonstrate this, we accumulate the attention computation latency on each GPU. The results have been given in Figure 4.

PP Level Imbalance: As shown in Figure 4 (a)(1), the attention computation latency across DP workers varies significantly. Within each DP worker, we observe "vertical lines" formed by the data points, each representing a PP worker within a DP worker. PP workers within the same DP worker exhibit nearly identical workloads, as they process the same

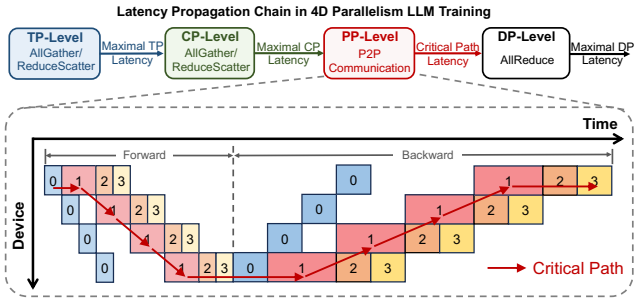


Figure 5: The process of latency propagation in 4D parallelism LLM training across different parallelism hierarchies. The impact of workload imbalance is enlarged at the PP level.

set of micro-batches. Based on the results shown in Figure 4 (a)(1), we conclude that the workload imbalance at both the DP and PP levels stems from the imbalance across micro-batches, which is caused by the input packing process. As illustrated in the left part of Figure 4 (b), input documents are packed into sequences (micro-batches) of equal length. While this fixed-length packing strategy ensures that each PP and DP worker processes the same number of tokens, variations in per-token computation intensity result in workload imbalances at these levels. For instance, a micro-batch containing only a single long document (highlighted in red) has larger workload compared to a micro-batch composed of multiple shorter documents (highlighted in green).

CP Level Imbalance: To better illustrate the imbalance at the CP level, we zoom in on a specific PP worker (also referred to as a CP group). As shown in Figure 4 (a)(2), significant workload imbalances are observed across CP workers (indicated by circles), while the TP workers within each CP worker exhibit similar computation latencies (data points within each circle). This imbalance across CP workers comes from the sequence sharding at the CP level. As shown in Figure 4 (b)(2), an input sequence is divided into chunks with an equal number of tokens, which are then distributed to CP workers. The state-of-the-art approach partitions the input sequence into $2 \times CP_size$ chunks. The i-th CP worker is assigned the i-th and $(2 \times CP_size - 1 - i)$-th chunks to achieve better load balancing [6]. This sharding strategy works well when the sequence only has a single document. However, if the sequence is packed with multiple documents, it may lead to significant workload imbalances across CP workers, as demonstrated in the figure. Although sequence chunks are further divided and distributed among TP workers, no imbalance is observed at the TP level. This is because, prior to computation, all TP workers perform an AllGather operation to collect the entire sequence chunk. As a result, each TP worker within a CP worker processes the same sequence chunk, eliminating imbalances at the TP level.

Imbalance Propagation: During training, the workload imbalance will be propagated from inner-level parallelism to

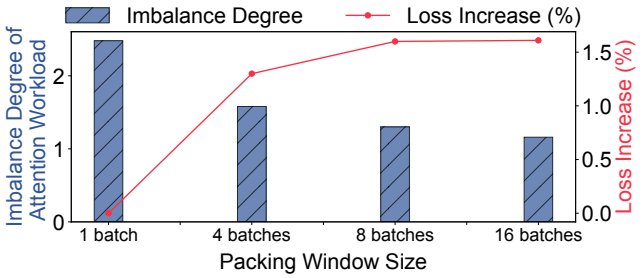


Figure 6: A larger packing window improves workload balance but leads to an increase in training loss.

outer-level parallelism. The imbalance will be accumulated and amplified, finally leading to a significant impact on end-to-end training latency. In DP, CP, and TP, collective communication [28] such as AllReduce, AllGather, and ReduceScatter is performed during training. All workers in these parallelism hierarchies work in a synchronized manner. As a result, the training latency of a given DP, CP, and TP group is determined by the slowest worker within that group. In contrast, at the PP level, different PP workers serve as producers and consumers of each other. As shown in Figure 5, the critical path of PP is the latency of the largest micro-batch traversing all PP workers plus the forward and backward passes of remaining micro-batches on the first PP worker. The distinct data dependency relationship between PP workers amplifies the imbalance, resulting larger impact on total latency. Due to the imbalance propagation, higher micro-batch training latency can arise from two main reasons: (1) Imbalances propagated from inner-level hierarchies (e.g., CP sharding imbalances); (2) Micro-batches with inherently larger workloads due to PP-level packing. This highlights the importance of eliminating the imbalance in all parallelism hierarchies.

3.2 Baseline Solution: Fixed-Length Packing

A potential solution to address the workload imbalance issue is to optimize the packing of input documents. Current 4D parallelism frameworks require all micro-batches to have a uniform sequence length [6], equal to the context window size, to enable efficient batching of input sequences. Building on this fixed-length requirement, we implement a baseline shuffling and packing optimization that packs input documents into micro-batches of equal size. Formally, given a set of input documents from one or more global batches, the objective is to pack these documents into several micro-batches with a fixed total length and balance the attention computation workload among all micro-batches. Without loss of generality, we use the causal mask as an example to calculate attention workload. With causal mask, the attention computation workload of a micro-batch is proportional to $\sum_{i=1}^{N} d_i^2$, where d_i is the length of each document within the micro-batch. The problem of searching the optimal document packing is NP-hard, as it can be extended from the classic Number Partition Problem [10] by adding a number sum constraint and a square-sum objective. To search for the optimal packing, we formulate the task as an integer linear programming (ILP) problem. Assuming we have N documents, each has length d_i, and we would like to pack these documents into M micro-batches, each with a total length of S. The objective is to minimize the maximum workload among the micro-batches:

$$\begin{aligned} \text{minimize} \quad & \max(\sum_{i=1}^{N} x_{ij} \cdot d_i^2),\ j = 1, \cdots, M \\ \text{subject to} \quad & \sum_{j=1}^{M} x_{ij} = 1, \qquad i = 1, \cdots, N \\ & \sum_{i=1}^{N} x_{ij} \cdot d_i \le S, \qquad j = 1, \cdots, M \\ & x_{ij} \in \{0, 1\} \end{aligned} \tag{1}$$

in which x_{ij} is a binary variable representing the packing plan. Specifically, $x_{ij} = 1$ means document i is packed into micro-batch j. With this ILP formulation, we then use a commercial solver [8] to obtain the optimal packing plan.

3.3 Tradeoff Analysis

Optimizing input document packing across more global batches could help achieve a higher degree of workload balance. However, it also manipulates the execution order of more input documents, affecting the randomness of data sampling and loading. This may negatively impact model quality and affect model convergence. To evaluate the tradeoff between packing balance and model quality, we pretrain a 550M-parameter model for 52K steps using various packing window sizes. We then assess the degree of workload imbalance of input batches after packing under different settings. The imbalance degree is calculated as $\frac{Max_Attn}{Avg_Attn}$, where Max_Attn represents the maximum attention computation workload in the global batch, and Avg_Attn denotes the average attention computation workload of all micro-batches in the global batch. As shown in Figure 6, when optimizing the packing across a single global batch, the workload imbalance across micro-batches still remains high. If the number of global batches increases, the fixed-length packing optimization could achieve a better workload balance. However, the final training loss increases as more global batches are involved in the packing optimization, due to reduced data loading randomness caused by repacking a larger number of input documents. These results indicate that naïve fixed-length packing optimization cannot achieve a good workload balance without compromising model quality, highlighting the need for more advanced solutions.

The tradeoff between packing balance and model quality motivates us to break the fixed-length constraint of micro-batches and design a more flexible packing strategy. In the following two sections, we will present the details of *WLB-LLM*,

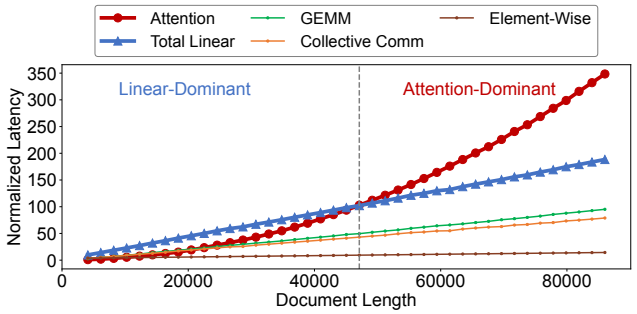


Figure 7: The relationship between operation latency and the input document length. (Total Linear is the sum of GEMM, collective communication, and element-wise operators)

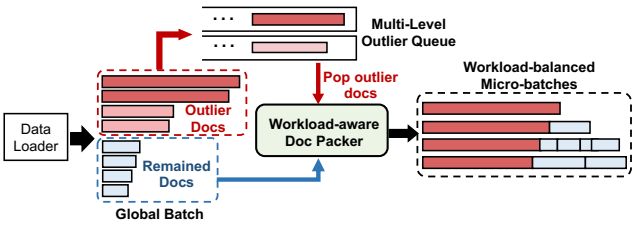


Figure 8: The process of outlier document delay combined with var-length packing.

including the PP-level variable-length packing and heuristic outlier document delay optimization (§4) and the CP-level fine-grained and adaptive sharding optimization (§5).

4 Var-Len Packing and Outlier Delay for PP

At the PP level, we focus on balancing the workload across micro-batches by repacking input documents in a workload-aware manner. **First**, we design a variable-length packing strategy to achieve a higher degree of workload balance within the same packing window size compared to fixed-length packing (§4.1). **Second**, we propose an outlier document delay method to adaptively delay the training of extremely long documents. This approach helps minimize the impact on data randomness while achieving near-optimal workload balance across micro-batches (§4.2). **Finally**, we design and implement an efficient heuristic algorithm to optimize packing at runtime with negligible overhead (§4.3).

4.1 Workload-Aware Var-Length Packing

The main limitation of the baseline fixed-length packing is that it cannot achieve balance if there is an extremely long document in a global batch. For example, if the length of a document equals the context window size, it becomes impossible to create another micro-batch consisting of shorter documents with an equal computation workload due to the quadratic complexity of attention computation. To address this limitation, we propose a variable-length packing strategy, which allows each micro-batch to have a different sequence length. The key insight behind our design is that the workload of a micro-batch is not solely determined by attention computation. Other operations, such as GEMM computation, element-wise operations, and collective communication (e.g., AllGather and ReduceScatter), also contribute significantly to the training latency and are influenced by the documents within each micro-batch. To demonstrate it, we present the relationship between operation latency and document length in Figure 7. The operation latency is measured from a training job of Llama2 7B model on 16 H100 GPUs and is normalized to the attention computation latency at a document length of 4096. It can be observed that attention computation latency increases quadratically with the length of input documents, while other operations, such as GEMM, collective communication, and element-wise operations, exhibit a linear relationship between operation latency and document length.

This relationship presents an opportunity to further improve workload balance beyond fixed-length packing. If a long document has significantly higher attention computation latency compared to other operations, we can pack multiple shorter documents together to extend the latency of other operations, thereby matching the total latency of the long document. Specifically, we extend the fixed-length packing to a variable-length approach. The optimization goal shifts from balancing only the attention computation workload to balancing the total workload, including all operations:

$$\begin{aligned} \text{minimize} \quad & \max\Big(\sum_{i=1}^{N}\big(W_a(x_{ij}\cdot d_i)+W_l(x_{ij}\cdot d_i)\big)\Big), \\ & \qquad\qquad\qquad\qquad j=1,\cdots,M \\ \text{subject to} \quad & \sum_{j=1}^{M} x_{ij}=1, \qquad\qquad i=1,\cdots,N \\ & \sum_{i=1}^{N} x_{ij}\cdot d_i \le S_{max}, \qquad j=1,\cdots,M \\ & x_{ij}\in\{0,1\} \end{aligned} \tag{2}$$

in which $W_a(\cdot)$ and $W_l(\cdot)$ are latency prediction functions that compute the attention computation latency and the latency of all other operations, respectively, based on document length. Both $W_a(\cdot)$ and $W_l(\cdot)$ can be derived from offline profiling. And S_{max} represents the maximum sequence length permitted by GPU memory constraints.

4.2 Outlier Document Delay

Our variable-length packing strategy enables a higher degree of balance by allowing short documents to be packed into a longer sequence that exceeds the fixed context window size. However, within a single global batch, the number of short sequences may be insufficient to fully balance the computation

workload across all micro-batches. To address this challenge, we propose adaptively delaying the execution of *outlier documents*, i.e., extremely long documents. This approach is based on our observation that, although these outlier documents have a significant impact on workload imbalance, they only contribute a small proportion of tokens for training (as described in §2.2). By selectively delaying the training of tokens from a few extremely long documents, we could achieve a more balanced workload across micro-batches while minimizing the impact on data randomness.

Overall Workflow: As shown in Figure 8, we implement a *multi-level waiting queue* for outlier documents. Assuming we have n outlier queues, each queue is associated with a hyperparameter L_i, which specifies the minimum length of documents in the i-th queue, where $L_1 < L_2 < \cdots < L_{n+1}$. When a new global batch arrives, documents with lengths greater than L_1 are considered as outliers and are added to the corresponding waiting queue i, where $L_i \leq d < L_{i+1}$, and d denotes the document length. The execution of documents in the outlier queue is delayed until the queue accumulates enough outlier documents. Once the queue size reaches the number of micro-batches in a global batch, the outlier documents are popped from the queue and added to each micro-batch. This ensures workload balance across micro-batches, as each micro-batch receives an equal number of outlier documents.

Tuning Hyperparameter L_i: The choice of hyperparameters L_i directly influences performance. A smaller gap between two consecutive hyperparameters L_i and L_{i+1} helps achieve a higher degree of workload balance, as it ensures that documents within the same queue have smaller differences in length. However, a smaller gap also causes documents within each queue to wait longer before accumulating enough documents to be processed together, thereby increasing the per-token delay. To select appropriate values for L_i, we sample a small subset of training documents and evaluate the packing algorithm (described later in §4.3) on this subset by measuring both the achieved workload balance across micro-batches and the resulting per-token delay. We then choose the optimal L_i values that maximize workload balance while maintaining a low per-token delay.

4.3 Heuristic Packing Algorithm

Although an ILP solver can derive the optimal packing for a given set of input documents, its solving time is impractically high to be paid at runtime. Optimizing document packing offline is also infeasible, as it would introduce significant storage overhead to store pre-packed datasets for different training configurations. To address this, we design a heuristic algorithm that combines variable-length packing with outlier document delay optimization to efficiently produce packed micro-batches with balanced workloads. As listed in Algorithm 1, the algorithm takes a dataloader D, a multi-level waiting queue Q for outlier documents operating in a FIFO manner, the number of micro-batches per iteration N, and the upper bound for sequence length S_{max} as input. The algorithm outputs a series of packed input batches with balanced workloads for each training iteration. The packing process begins by iterating over the input batches in the dataloader D and adding all outlier documents into the corresponding waiting queue in Q for delayed processing (Line 4-10). If any queue q in Q reaches size N, the documents in q are popped out and added to the pending document set for the current batch (Line 11-15). Packing proceeds in a greedy manner: it first processes the remaining documents, followed by the newly added long documents. For each document, the algorithm first attempts to pack it into the micro-batch with the minimal workload (calculated using $W_a(\cdot)$ and $W_l(\cdot)$ as defined in Equation 2), provided that the total length remains within the upper bound S_{max}. If this is not feasible, it then

Algorithm 1: Heuristic Var-length Packing Algorithm

```
   input : Dataloader: D, Multi-level waiting queue: Q,
           The number of micro-batches per iteration: N,
           Sequence length upper bound: S_max
   output: Packed input batches for training: B
1  Remained_Doc = [];
2  for Cur_Batch in D do
3      New_Docs = [];
4      for Doc in Cur_Batch do
           /* Delay the execution of outlier documents.          */
5          if Doc.Is_Outlier() then
6              Q.Add(Doc);
7          else
8              New_Docs.Push(Doc);
9          end
10     end
11     for q in Q do
12         if len(q) ≥ N then
               /* Pop outlier documents for the current batch.   */
13             New_Docs.Push(q.Pop(N));
14         end
15     end
       /* Sort the documents in descending order by length.      */
16     New_Docs.Sort_by_Length();
       /* Packing remaining documents first.                     */
17     Doc_Set = Remained_Doc + New_Docs
18     Remained_Doc.Clear();
19     New_Batch = Create_Batch(N);
20     for Doc in Doc_Set do
           /* Get micro-batches with minimum workload/length.    */
21         W_idx = New_Batch.Get_Min_Workload();
22         L_idx = New_Batch.Get_Min_Length();
23         if New_Batch[W_idx].Len() + Doc.Len() < S_max then
24             New_Batch[W_idx].Push(Doc);
25         else
26             if New_Batch[L_idx].Len() + Doc.Len() < S_max then
27                 New_Batch[L_idx].Push(Doc);
28             else
29                 Remained_Doc.Push(Doc);
30             end
31         end
32     end
33     B.Push(New_Batch);
34 end
```

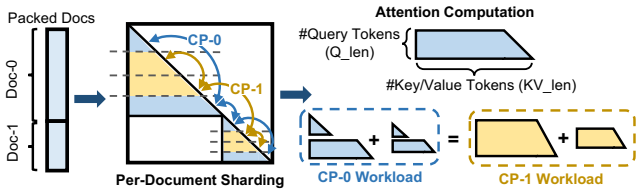


Figure 9: Overview of fine-grained per-document sharding.

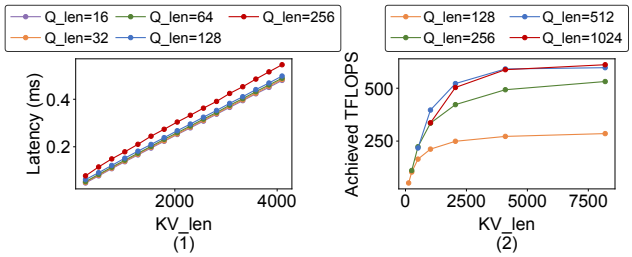


Figure 10: Attention kernel performance profiling: (Left) Attention forward latency; (Right) Achieved TFLOPs of the attention forward kernel.

attempts to pack the document into the micro-batch with the minimal current length (Lines 20-27). Documents that cannot fit within these constraints are saved for the next iteration (Line 29). Our heuristic approach effectively combines outlier document delay and variable-length packing to efficiently balance computational workloads across all micro-batches.

5 Fine-grained and Adaptive Sharding for CP

At the CP level, we aim to improve workload balance across document shards by implementing a fine-grained per-document sharding strategy, ensuring that each CP worker receives an equal computation workload (§5.1). Additionally, we observe a tradeoff between attention kernel efficiency and sharding granularity (§5.2). To maximize overall performance, we conduct an in-depth analysis and adaptively select the optimal sharding strategy for a given input sequence (§5.3).

5.1 Per-Document Sharding Design

At the CP level, the sequence of micro-batches is sharded across CP workers. Each CP worker works on an exclusive sequence shard. Existing CP implementation employs a **Per-Sequence Sharding** strategy, which equally shards the entire input sequence into $2 \times CP_size$ chunks. This method could easily lead to significant attention computation workload imbalance when the input sequence is packed with multiple documents. To eliminate the workload imbalance issue at the CP level, we propose sharding the sequence in a fine-grained manner. Specifically, we conduct **Per-Document Sharding** to divide each document into $2 \times CP_size$ document chunks. As shown in Figure 9, each CP worker takes a symmetrical pair of document chunks for each input document. With our fine-grained per-document sharding strategy, each CP worker not only receives the same number of tokens (ensuring workload balance in GEMM computation and collective communication) but also gets the same attention computation workload.

Avoid Padding: Our fine-grained per-document sharding strategy divides each input document into $2 \times CP_size$ document chunks. However, document lengths are not always divisible by $2 \times CP_size$, requiring extending the document length through adding some padding tokens. To avoid the redundant computation introduced by document padding, we design a padding-free per-document sharding method. Specifically, we split each document into two parts: one divisible by $2 \times CP_size$ and the remaining tokens. Assuming the length of the i-th document is d_i, where $d_i = e_i + r_i$, with $e_i = \lfloor \frac{d_i}{2 \times CP_size} \rfloor$. We apply the standard per-document sharding on the e_i part, while the tokens in r_i are distributed to CP workers in a round-robin fashion. Since $\sum_{i=1}^{n} d_i$ and $\sum_{i=1}^{n} e_i$ are both divisible by $2 \times CP_size$, it follows that $\sum_{i=1}^{n} r_i = \sum_{i=1}^{n} (d_i - e_i)$ is also divisible by $2 \times CP_size$. This ensures that each CP worker receives an equal number of tokens, thereby eliminating the need for padding.

5.2 Kernel Efficiency vs. Sharding Balance

Our per-document sharding strategy fully eliminates workload imbalance at the CP level. However, splitting each document into multiple shorter chunks may compromise kernel efficiency. The kernel efficiency may drop with fine-grained sharding due to two major reasons: **(1) Tile-level Computation Wasting:** The computation of attention is split into smaller tiles and distributed to different thread blocks for execution on GPU. For example, in the attention forward kernel of FlashAttention [5], the tile size is set to 128. If the number of tokens is less than the tile size, the thread block will still perform the full computation on 128 tokens, which will waste a significant amount of computation. To illustrate this, we profile the attention forward latency for query token lengths ranging from 16 to 256. As shown in Figure 10 (Left), when the number of query tokens (Q_len) increases from 16 to 128, the kernel latency remains constant. This is because all short documents with fewer than 128 tokens are padded to 128 tokens for computation at the kernel level. In contrast, as Q_len increases from 128 to 256, the kernel latency rises significantly. **(2) Inefficient Tensor Memory Accelerator (TMA) Usage:** TMA is a feature introduced in the NVIDIA Hopper architecture that enables asynchronous memory copying between global memory and shared memory on GPUs [1]. With large document lengths (e.g., $Q_len \geq 256$), multiple thread blocks process different Q tokens while sharing the same KV tokens of the document chunk. This allows KV tensor loading to be shared via the L2 cache using TMA load multicast,

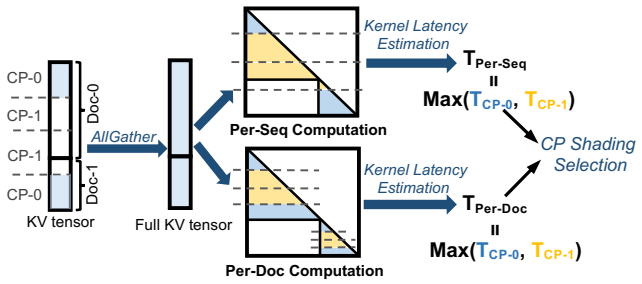


Figure 11: The process of adaptive CP sharding selection.

Model Size	Context Window	#GPU	4D Parallelism Configs (TP, CP, PP, DP)
550M	*64K*	*32*	*(2, 2, 4, 2)*
	128K	*32*	*(2, 4, 4, 1)*
7B	*64K*	*32*	*(4, 2, 4, 1)*
	128K	*64*	*(8, 2, 4, 1)*
30B	*64K*	*64*	*(8, 2, 4, 1)*
	128K	*128*	*(8, 4, 4, 1)*
70B	*64K*	*256*	*(16, 4, 4, 1)*
	128K	*256*	*(16, 4, 4, 1)*

Table 1: Model and 4D parallelism configurations.

significantly reducing the latency of transferring KV tensors from global memory to shared memory. As shown in Figure 10 (Right), when the length of the Query tensor increases from 128 to 256, the achieved TFLOPs increase significantly, demonstrating the impact of leverage TMA load multicast.

These profiling results clearly demonstrate the tradeoff between attention kernel efficiency and CP sharding balance. If per-document sharding is applied to input sequences composed of short documents, it can introduce redundant computation at the kernel level and reduce the achieved TFLOPs, ultimately leading to longer attention computation latency.

5.3 Adaptive Sharding Selection

Based on our analysis in Section 5.2, although fine-grained per-document sharding achieves optimal workload balance at the CP level, it does not necessarily guarantee better performance, as the attention kernel may become less efficient with the more fine-grained document chunks generated by per-document sharding. To address this issue, we propose to adaptively select the optimal sharding strategy for each micro-batch at runtime. As shown in Figure 11, during the forward pass at the CP level, an AllGather communication is performed across CP workers to collect the full KV tensor. We then calculate the input tensor shapes for the attention kernel (number of query tokens and key/value tokens) in the per-sequence and the per-document sharding cases. Finally, we predict the attention kernel latency and select the CP sharding strategy that yields lower attention computation latency.

To accurately estimate the attention kernel latency, we leverage the insights provided in §5.2. First, we calculate the total floating point operations required for the attention computation. The kernel-level padding is also considered by padding the document chunk to a multiple of the tiling size. Next, we estimate the achieved TFLOPs for the given tensor shape using the data collected from offline profiling which includes the impact of TMA usage. Finally, the attention kernel latency is estimated by dividing the amount of floating point operations by the achieved TFLOPs. By adaptively selecting CP sharding, *WLB-LLM* minimizes the CP level training latency.

6 Implementation Detail

WLB-LLM is built on top of a widely adopted 4D parallelism training paradigm, which integrates advanced distributed training techniques at each parallelism level. For DP, *WLB-LLM* leverages Fully Sharded Data Parallel (FSDP) [54], a state-of-the-art approach that uniformly shards model parameters across DP workers, significantly reducing memory requirements compared to traditional DP methods. For PP, *WLB-LLM* employs the interleaved 1F1B pipeline schedule [26]. To support variable-length packing optimization, *WLB-LLM* further implements a variable-length pipeline, which enables micro-batches to have variable sequence lengths. For CP, the per-sequence sharding baseline follows the AllGather-based CP approach used in Llama3 training [6], which performs an AllGather communication during the forward pass across all CP workers to collect the full KV tensors for attention computation. Building on this, *WLB-LLM* implements fine-grained per-document sharding by optimizing the document partitioning and distribution. For TP, *WLB-LLM* performs 1D tensor parallelism with sequence parallelism enabled [15, 38]. Additionally, *WLB-LLM* incorporates computation–communication overlapping to further improve TP training performance [46].

7 Evaluation

In this section, we evaluate *WLB-LLM* across various LLM sizes and 4D parallelism configurations, with model sizes ranging from *550M* to *70B*. We begin by presenting the improvements in end-to-end training latency. Next, we analyze the speedup contributions from individual optimizations at the PP level (§4) and the CP level (§5), respectively. Finally, we demonstrate that the optimizations of *WLB-LLM* do not compromise model quality or model convergence by comparing the training loss curve.

7.1 Experiments Setup

Hardware: We deploy *WLB-LLM* on a cluster with 32 nodes. Each node is equipped with 8× NVIDIA H100 SXM 80GB GPUs interconnected via high-bandwidth NVLink, while

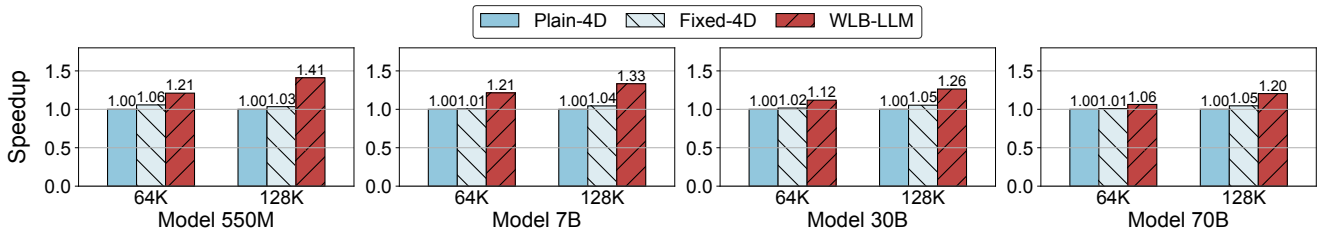


Figure 12: Training performance speedups of *WLB-LLM* and *Fixed-4D* over *Plain-4D* across various configurations.

cross-node communication is facilitated by RDMA over Converged Ethernet (RoCE).

Models and Parallelism Configurations: We conduct experiments on a series of our internal LLaMA-like models, spanning four different scales: 550*M*, 7*B*, 30*B*, and 70*B*. The 7*B* model shares the same architecture as the LLaMA2-7B model [42]. The other models retain the same architecture while proportionally adjusting the number of layers and model dimension size. For each model, we evaluated performance using two different context window sizes: 64*K* and 128*K*. Each model scale and context window size is associated with a corresponding 4D parallelism configuration. The global batch size is set to $PP_size \times DP_size$ and we use *bfloat16* precision for all evaluations. Details of the training setup and parallelism configurations are provided in Table 1. When mapping 4D parallelism to the hardware, inner-level parallelism dimensions (e.g., TP or CP) are prioritized for mapping to intra-node GPUs, leveraging the high-bandwidth NVLink for efficient communication. Outer-level parallelism dimensions, such as DP, are subsequently mapped across multiple nodes. Throughout the rest of the paper, we use *Model Size-Context Window Size* to denote a specific configuration. For instance, *7B-128K* refers to the 7*B* model with a 128*K* context window.

Baselines: We compare *WLB-LLM* with two baselines:

- **Plain-4D**: This is our internal codebase for large-scale LLM training, supporting 4D parallelism to enable efficient training and scaling up to 100*K* GPUs. *Plain-4D* directly uses input batches obtained from the dataloader for training without optimizing the input packing. For CP sharding, *Plain-4D* employs a per-sequence sharding method, which shards the input sequence at the whole-sequence level.

- **Fixed-4D**: *Fixed-4D* applies the baseline fixed-length packing optimization, as described in Section 3.2. To minimize packing overhead, a greedy algorithm is used instead of the solver, and the packing size is restricted to a single global batch to preserve data loading randomness and prevent an increase in training loss. For CP sharding, *Fixed-4D* utilizes a fixed sharding strategy throughout the entire training process, either per-sequence or per-document. We then select the better-performing result between the two sharding strategies and use it for comparisons.

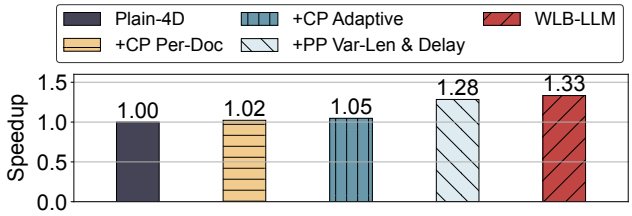


Figure 13: Performance breakdown of *WLB-LLM* on the 7B model with a 128K context window.

7.2 Training Performance

We run *WLB-LLM* and all baselines on diverse model size and context window size:

Plain-4D vs. Fixed-4D: As shown in Figure 12, *Fixed-4D* achieves only marginal improvements over *Plain-4D*, with an average speedup of approximately 1.03× across all settings. This limited gain is primarily because *Fixed-4D* adjusts document packing only within a single global batch and is constrained by the context window size. It fails to address the presence of outlier documents with extremely long lengths (e.g., a document with a length equal to the context window size). Given that attention computation scales quadratically with document length, packing multiple short documents within the constraints of the context window cannot match the computation workload of an extremely long document. Moreover, *Fixed-4D* employs either per-sequence or per-document sharding at the CP level for all input batches. This approach overlooks the tradeoff between attention kernel efficiency and sharding balance, resulting in suboptimal performance. These limitations constrain the improvements of *Fixed-4D* over the *Plain-4D* baseline.

***WLB-LLM* vs. Baselines**: Figure 12 shows that *WLB-LLM* consistently outperforms all baselines across various model sizes and 4D parallelism configurations. Specifically, *WLB-LLM* achieves speedups of 1.23× and 1.19× over *Plain-4D* and *Fixed-4D*, respectively. The significant improvement stems from two key optimizations in *WLB-LLM*. At the PP

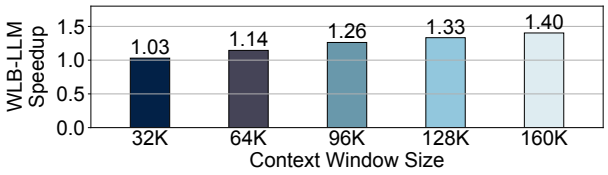


Figure 14: Speedups of *WLB-LLM* on the 7B model across context window sizes.

level, *WLB-LLM* employs a heuristic variable-length document packing algorithm that is not constrained by the fixed context window size. This algorithm also selectively delays the training of outlier documents, achieving a higher degree of workload balance compared to fixed-length packing. At the CP level, *WLB-LLM* combines per-sequence and per-document sharding, adaptively selecting the optimal sharding strategy for each micro-batch to maximize the performance.

Across Model Size and Context Window Size: As shown in Figure 12, *WLB-LLM* achieves slightly lower speedup on larger models. This is because larger models involve more GPUs for training, which increases the ratio of communication latency to computation latency, making the impact of workload imbalance in the attention layer less significant. On the other hand, increasing the context window size from 64K to 128K improves the average speedup from 1.15× to 1.30×, as longer contexts exacerbate workload imbalance issue. A more detailed sensitivity analysis on the context window size is provided in Section 7.3.

7.3 Breakdown and Sensitivity Analysis

In this section, we conduct a performance breakdown and sensitivity analysis. Specifically, we first report the performance breakdown to evaluate the impact of each optimization technique. Then, we perform a sensitivity study by measuring the speedups across different context window sizes.

Speedup Breakdown: We show the speedup breakdown in Figure 13 by separately applying each optimization technique proposed in *WLB-LLM* to *Plain-4D* for the *7B-128K* training configuration. By utilizing the fine-grained per-document sharding at the CP level, we observe a 1.02× speedup due to the reduced imbalance among CP workers. The speedup is limited due to the tradeoff between the kernel efficiency and the sharding balance. As a result, performing per-document sharding is not always beneficial and can potentially cause performance degradation. By adaptively selecting the per-document and per-sequence sharding strategies, we can improve the speedup to 1.05×. We then apply the PP-level optimizations on *Plain-4D* to study their effect. It can be observed that combining the heuristic var-length packing with the outlier documents delay offers a significant speedup of 1.28×. Finally, we incorporate both CP and PP-level optimizations to maximally reduce the workload imbalance across the all

Packing Method		Imbalance Degree	Packing Overhead (ms)
Method	**Config**		
Original Packing	/	1.44	0
Fixed-Len Greedy	#global batch=1	1.41	4
	#global batch=2	1.22	5
	#global batch=4	1.11	5
	#global batch=8	1.08	5
Fixed-Len Solver	#global batch=1	1.40	467
	#global batch=2	1.18	1488
	#global batch=4	1.09	25313
WLB-LLM	#queue=1	1.24	8
	#queue=2	1.05	20
	#queue=3	1.05	23

Table 2: Packing imbalance degree and overhead analysis.

parallelism hierarchies, leading to a final speedup of 1.33×.

Speedup across Context Window Sizes: We investigate the impact of context window size on the performance improvements delivered by *WLB-LLM*. Figure 14 shows the speedups over *Plain-4D* on 7*B* model across different context window sizes, varying from 32*K* to 160*K*. We observe that as the context window size grows, the achieved speedup becomes more significant, reaching 1.40× with a 160K context window. This is because a larger context window raises the likelihood of outlier documents appearing. Additionally, a larger context window also increases the proportion of attention computation and exacerbates the impact of imbalance on training latency. The trend of increasing speedups demonstrates the significant potential of *WLB-LLM* in handling the ever-expanding context window sizes today.

7.4 Optimization Analysis

In this section, we first analyze the effectiveness of the packing and sharding optimization in *WLB-LLM*. Furthermore, we demonstrate that the system optimizations in *WLB-LLM* do not compromise model quality or slow down convergence.

Packing Balance and Overhead Analysis: To assess the balance degree of computation workload across micro-batches with different packing strategies, we profile and compare the forward latency of each micro-batch in a *7B-128K* training job with different packing methods and configurations applied. As discussed in Section 3.1, the PP level latency is primarily determined by the largest micro-batch. Therefore, we use the following metric to represent the imbalance degree of a given batch: $\frac{Max_Latency \times PP_size}{Total_Latency}$, where $Max_Latency$ is the forward latency of the largest micro-batch and $Total_Latency$ is the total forward latency of all micro-batches. A lower imbalance degree indicates that the given batch is more balanced in terms of workload. The result of the imbalance degree and the packing overhead (per-batch packing latency) of different methods have been given in Table 2. We evaluate four different packing methods: (1) *Original Packing*, which uses the original input batch loaded from the dataloader; (2) *Fixed-Len Greedy* is the packing method used in *Fixed-4D* baseline,

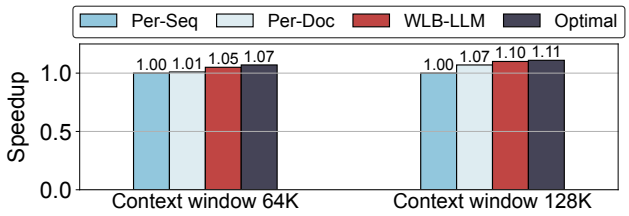


Figure 15: CP sharding performance comparison.

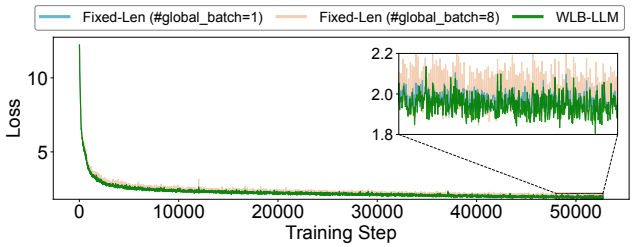


Figure 16: Training loss comparison on a *550M* model.

which shuffles documents in several global batches in a greedy manner to optimize workload balance across micro-batches; (3) *Fixed-Len Solver*, which employs an ILP solver [8] to solve Equation 1 and provides the optimal packing based on the given global batches; and (4) *WLB-LLM*, which utilizes var-len packing combined with outlier document delay. Additionally, we demonstrate the performance of *WLB-LLM* with different numbers of outlier document queues.

As shown in Table 2, *Fixed-Len Greedy* can slightly mitigate workload imbalance when packing across a single global batch. Packing across multiple global batches helps to achieve lower imbalance degrees, while it will incur higher training losses, as demonstrated in Figure 6. As for *Fixed-Len Solver*, it achieves lower imbalance degrees compared to *Fixed-Len Greedy* under the same number of global batches. However, the solver-based solution suffers from significant packing overhead. For instance, when packing across 4 global batches, the average packing latency for each batch exceeds 25 seconds. In contrast, *WLB-LLM* is the only solution which achieves both near-optimal imbalance degree and low packing overhead. For example, when having two outlier queues, *WLB-LLM* achieves 1.05 imbalance degree. Additionally, the per-batch packing latency is only 20 ms which is less than 0.65% when compared to the per-step training latency.

CP Sharding Performance Analysis: To demonstrate the effectiveness of per-document sharding and the adaptive sharding selection optimization at the CP level in *WLB-LLM*, we conduct a case study on a single transformer layer of a *7B* model with CP size of 4. We compare the forward and backward latency with different sharding strategies including: (1) *Per-Sequence Sharding (Per-Seq)*, (2) *Per-Document Sharding (Per-Doc)*, (3) *WLB-LLM*, which determines the sharding between *Per-Seq* and *Per-Doc* adaptively based on the given micro-batch at runtime, and (4) *Optimal*, which is the optimal result. It always chooses the sharding from *Per-Seq* and *Per-Doc* that yields lower latency.

As shown in Figure 15, *Per-Document Sharding* achieves a speedup of 1.01× and 1.07× over the *Per-Sequence Sharding* baseline under context window sizes of 64K and 128K, respectively. These results demonstrate the effectiveness of *Per-Document Sharding* in reducing the workload imbalance at the CP level. However, the fine-grained *Per-Document Sharding* may sacrifice kernel efficiency, particularly when the input sequence consists of many short documents. To overcome this issue, *WLB-LLM* leverages an adaptive sharding selection method to intelligently choose a better sharding strategy for each micro-batch at runtime. The results in Figure 15 show that *WLB-LLM* achieves a 7.5% and 3.4% improvement over static *Per-Seq* and *Per-Doc* sharding, respectively. Furthermore, *WLB-LLM* is very close to the optimal result, demonstrating the effectiveness of our adaptive selection method.

Model Convergence Analysis: *WLB-LLM* employs a heuristic variable-length packing optimization that adjusts document packing and delays the execution of outlier documents. To demonstrate that this optimization does not affect model convergence, we present the training loss curve of our *550M* model. As shown in Figure 16, packing across 8 global batches results in a noticeable increase in training loss (1.6% on average). This is because packing over multiple global batches disrupts the randomness of data sampling in the dataloader, which leads to a different data distributed per batch than desired. In contrast, *WLB-LLM* follows almost the same trend as fixed-length packing across a single global batch, since the heuristic variable-length packing algorithm used by *WLB-LLM* only delays outlier documents, which contributes to a small proportion of all input tokens. According to our profiling, each token is delayed by an average of 0.5 iterations in *WLB-LLM*. This minimal delay preservers original data distribution at best, allowing *WLB-LLM* to improve training efficiency without compromising model quality.

8 Discussion

In this section, we discuss the compatibility of *WLB-LLM* with other parallelism dimensions and potential opportunities for improvement in future work.

Compatibility with Expert Parallelism: Beyond 4D parallelism, expert parallelism (EP) [20, 53] is a complementary dimension tailored for Mixture-of-Experts (MoE) models [37]. In EP, different experts are distributed across workers, and each input token is independently routed to one or more experts at runtime, which can introduce load imbalance when expert capacity is exceeded [50]. Prior works mitigate this by adding a load-balancing auxiliary loss [7, 20] or injecting expert-wise bias into the routing scores [45]. Thanks to

these optimizations, state-of-the-art MoE training achieves dropless token routing [22], guaranteeing that every token is dispatched strictly according to its gating logits without any capacity-driven drops. Consequently, the packing and sharding optimizations in *WLB-LLM* will not affect EP routing decisions, making *WLB-LLM* fully compatible with existing EP load-balancing strategies.

Further Optimization Opportunity: The adaptive CP sharding selection in *WLB-LLM* is performed at the granularity of the entire input sequence, where either per-sequence sharding or per-document sharding is applied to the whole sequence. Although this approach already achieves near-optimal performance, there remains potential for further optimization. In certain cases, such as when an input sequence contains both extremely long documents and many short documents, it may be beneficial to combine both sharding strategies and apply them selectively based on document lengths. For instance, per-document sharding could be applied to long documents to balance the workload, while per-sequence sharding could be applied to short documents to maximize attention kernel efficiency. We hope this observation could motivate future work to further improve the efficiency of CP training.

9 Related Work

Distributed LLM Training Frameworks: To address the challenges of training extra-large LLMs, prior approaches primarily rely on 3D parallelism [15, 26, 38, 40, 43, 44, 55], which integrates tensor parallelism [15, 17, 38], pipeline parallelism [11, 19, 21, 26], and data parallelism [35, 36, 54]. Although 3D parallelism has proven its effectiveness in scaling model size, enabling the training of models with trillions of parameters [40], it struggles to scale context window size effectively. To address this limitation, a new dimension of parallelism called context parallelism has been introduced [30], forming the 4D parallelism training paradigm. Context parallelism splits input sequences into chunks, effectively reducing the memory bottleneck associated with extremely long documents. Initially, context parallelism employs a ring-based approach that overlaps communication and computation using P2P communication [24]. More recent approaches leverage collective communication methods (e.g., AllGather or AllToAll) to aggregate key and value tensors, offering greater flexibility and better support for various types of attention masks [6, 12, 30]. The introduction of context parallelism enables efficient training of LLMs with long context windows. However, existing 4D parallelism frameworks overlook the heterogeneity in per-token arithmetic intensity, leading to significant workload imbalances across GPUs.

Input Padding and Packing for LLM Training: The input of LLMs consists of samples with varying lengths. To batch input documents together and optimize GPU utilization during LLM training, the input documents must be organized into tensors of identical lengths. This can be achieved through two primary approaches: *Padding* [38] and *Packing* [16]. Padding involves adding zero padding to shorter documents within a micro-batch. However, this approach inevitably introduces redundant computation, communication, and memory overhead. To address this issue, prior works have designed more efficient kernels to reduce redundant computation [49, 51]. Other works, such as DynaPipe [13], optimize batching strategies to minimize padding. Although these approaches mitigate redundant computation, the overhead caused by padding cannot be fully eliminated. To further address this problem, recent works propose packing short documents together to form a single long input sequence [16, 47]. After packing, an additional attention mask must be applied to ensure tokens only attend to others within the same document [6, 18, 29, 34]. Due to its efficiency in avoiding redundant computation, packing has become the mainstream choice in LLM training. For example, LLaMA3 adopts input packing [6] in its training process, and state-of-the-art high-performance attention implementations (e.g., FlashAttention [4, 5]) also support efficient attention computation with document packing. *WLB-LLM* focuses on input packing and addresses the workload imbalance issue that arises when input documents are packed together.

10 Conclusion

In this paper, we present *WLB-LLM*, a workload-balanced 4D parallelism framework designed to enhance the efficiency of LLM training. *WLB-LLM* systematically identifies and addresses workload imbalances that arise across multiple parallelism hierarchies. At the pipeline parallelism level, *WLB-LLM* introduces a novel heuristic variable-length document packing algorithm combined with an outlier document delay method that effectively mitigates workload imbalance across micro-batches. At the context parallelism level, *WLB-LLM* proposes a fine-grained per-document sharding method with an adaptive sharding selection strategy to maximize CP training efficiency. Comprehensive experiments demonstrate that *WLB-LLM* outperforms existing 4D parallelism frameworks across various model sizes and parallelism configurations, achieving an average speedup of 1.23×.

11 Acknowledgments

We would like to express our sincere appreciation to the anonymous reviewers and our shepherd for their insightful feedback and invaluable suggestions, which played a significant role in shaping and improving our final paper. We also thank the Meta AI and Systems Co-Design team for their continuous support and collaboration, which were crucial to the success of this project. Work at the University of California, San Diego was supported in part by NSF grant 2124039.

References

[1] H100 GPU. https://www.nvidia.com/en-us/data-center/h100/.

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

[3] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. *See https://vicuna.lmsys.org (accessed 14 April 2023)*, 2(3):6, 2023.

[4] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[5] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*, 2024.

[7] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[8] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[9] Eric Harper, Somshubra Majumdar, Oleksii Kuchaiev, Li Jason, Yang Zhang, E Bakhturina, V Noroozi, S Subramanian, K Nithin, H Jocelyn, et al. NeMo: A Toolkit for Conversational AI and Large Language Models. *Computer software], URL: https://github.com/NVIDIA/NeMo*, 2019.

[10] Juris Hartmanis. Computers and Intractability: A Guide to the Theory of NP-Completeness (Michael R. Garey and David S. Johnson). *Siam Review*, 24(1):90, 1982.

[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *Advances in neural information processing systems*, 32, 2019.

[12] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *arXiv preprint arXiv:2309.14509*, 2023.

[13] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. DynaPipe: Optimizing Multi-task Training through Dynamic Pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 542–559, 2024.

[14] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*, 2020.

[15] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing Activation Recomputation in Large Transformer Models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[16] Mario Michael Krell, Matej Kosec, Sergio P Perez, and Andrew Fitzgibbon. Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance. *arXiv preprint arXiv:2107.02027*, 2021.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, 25, 2012.

[18] Achintya Kundu, Rhui Dih Lee, Laura Wynter, Raghu Kiran Ganti, and Mayank Mishra. Enhancing training efficiency using packing with flash attention. *arXiv preprint arXiv:2407.09105*, 2024.

[19] Joel Lamy-Poirier. Breadth-First Pipeline Parallelism. *Proceedings of Machine Learning and Systems*, 5:48–67, 2023.

[20] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[21] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[22] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[23] Hao Liu and Pieter Abbeel. Blockwise Parallel Transformers for Large Context Models. *Advances in Neural Information Processing Systems*, 36, 2024.

[24] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring Attention with Blockwise Transformers for Near-Infinite Context. *arXiv preprint arXiv:2310.01889*, 2023.

[25] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[27] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[28] Nvidia. NVIDIA Collective Communication Library (NCCL). developer.nvidia.com/nccl.

[29] NVIDIA. NVIDIA NeMo Framework: Sequence Packing. https://docs.nvidia.com/nemo-framework/user-guide/latest/sft_peft/packed_sequence.html.

[30] NVIDIA. Megatron Core: Context Parallelism. https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html, 2023.

[31] OpenAI. Introducing ChatGPT. https://openai.com/index/chatgpt/, 2022.

[32] Myle Ott, Sam Shleifer, Min Xu, Priya Goyal, Quentin Duval, and Vittorio Caggiano. Fully Sharded Data Parallel: Faster AI Training with Fewer GPUs, 2021.

[33] Ray Perrault and Jack Clark. Artificial Intelligence Index Report 2024. 2024.

[34] Pytorch. FlexAttention: The Flexibility of PyTorch with the Performance of FlashAttention. https://pytorch.org/blog/flexattention/, 2024.

[35] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[36] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[37] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[39] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990*, 2022.

[40] DeepSpeed Team and Rangan Majumder. DeepSpeed: Extreme-Scale Model Training for Everyone, 2020.

[41] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805*, 2023.

[42] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2023.

[43] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin

Mohd-Yusof, et al. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[44] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. *arXiv preprint arXiv:2306.10209*, 2023.

[45] Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. Auxiliary-loss-free load balancing strategy for mixture-of-experts. *arXiv preprint arXiv:2408.15664*, 2024.

[46] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 93–106, 2022.

[47] Shuhe Wang, Guoyin Wang, Yizhong Wang, Jiwei Li, Eduard Hovy, and Chen Guo. Packing analysis: Packing is more appropriate for large models or datasets in supervised fine-tuning. *arXiv preprint arXiv:2410.08081*, 2024.

[48] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023.

[49] Jinle Zeng, Min Li, Zhihua Wu, Jiaqi Liu, Yuang Liu, Dianhai Yu, and Yanjun Ma. Boosting Distributed Training Performance of the Unpadded BERT Model. *arXiv preprint arXiv:2208.08124*, 2022.

[50] Zhiyuan Zeng, Qipeng Guo, Zhaoye Fei, Zhangyue Yin, Yunhua Zhou, Linyang Li, Tianxiang Sun, Hang Yan, Dahua Lin, and Xipeng Qiu. Turn waste into worth: Rectifying top-k router of moe. *arXiv preprint arXiv:2402.12399*, 2024.

[51] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 344–355. IEEE, 2023.

[52] Biao Zhang, Barry Haddow, and Alexandra Birch. Prompting Large Language Model for Machine Translation: A Case Study. In *International Conference on Machine Learning*, pages 41092–41110. PMLR, 2023.

[53] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *arXiv preprint arXiv:2502.19811*, 2025.

[54] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[55] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

A Artifact Appendix

Abstract

Our artifact provides an open-source implementation of the context parallelism optimization of *WLB-LLM*. It includes both the *Per-Sequence* baseline and the *Per-Document* sharding optimization proposed by *WLB-LLM*. This implementation differs from the internal version used at Meta. We develop this open-source version following the design proposed in *WLB-LLM*, using open-source frameworks. The goal is to make it easier for readers to experiment with and build upon the context parallelism optimizations presented in *WLB-LLM*.

Scope

This artifact provides a user-friendly interface to perform context parallelism training using both *Per-Sequence* and *Per-Document* sharding strategies. It enables quick performance comparisons between the two approaches, supporting the claims made in the *WLB-LLM* paper. It also serves as a good starting point for exploring further optimizations in context parallelism.

Contents

The repository includes the following components:

1. Implementation of both *Per-Sequence* and *Per-Document* sharding.
2. Code and scripts for verifying correctness.
3. Code and scripts for benchmarking the training efficiency of *Per-Sequence* vs. *Per-Document* sharding.
4. A detailed `README.md` with usage instructions and setup guidance.

Hosting

The artifact is open-sourced on GitHub at: https://github.com/Ash-Zheng/WLB-LLM-CP. We are actively maintaining the main branch of the repository to make it more comprehensive. Clear and accessible documentation will be continuously updated to reflect new features and improvements.

Requirements

We have tested and evaluated our artifact using the `nvcr.io/nvidia/pytorch:25.01-py3` Docker image, with PyTorch 2.6.0, FlashAttention 2.4.2, and CUDA 12.8. Our implementation has been validated on both NVIDIA A100 and H100 GPUs. To run performance comparisons between *Per-Sequence* and *Per-Document* sharding, a minimum of two GPUs is required.

DecDEC: A Systems Approach to Advancing Low-Bit LLM Quantization

Yeonhong Park* Jake Hyun* Hojoon Kim Jae W. Lee
Seoul National University

Abstract

Quantization of Large Language Models (LLMs) has recently gained popularity, particularly for on-device settings with limited hardware resources. While efficient, quantization inevitably degrades model quality, especially in aggressive low-bit settings such as 3-bit and 4-bit precision. In this paper, we propose DecDEC, an inference scheme that improves the quality of low-bit LLMs while preserving the key benefits of quantization: GPU memory savings and latency reduction. DecDEC stores the residual matrix—the difference between full-precision and quantized weights—in CPU, and dynamically fetches the residuals for only a small portion of the weights. This portion corresponds to the salient channels, marked by activation outliers, with the fetched residuals helping to correct quantization errors in these channels. Salient channels are identified *dynamically* at each decoding step by analyzing the input activations—this enables adaptation to the dynamic nature of activation distribution, thus maximizing the effectiveness of error compensation. We demonstrate the effectiveness of DecDEC by augmenting state-of-the-art quantization methods. For example, DecDEC reduces the perplexity of a 3-bit Llama-3-8B-Instruct model from 10.15 to 9.12—outperforming its 3.5-bit counterpart—while adding less than 0.0003% to GPU memory usage and incurring only a 1.7% inference slowdown on NVIDIA RTX 4050 Mobile.

1 Introduction

Recent advancements in Large Language Models (LLMs) based on the Transformer architecture [59] have shown great potential to reshape our daily lives [7,40,44,55,56]. However, their deployment costs pose a significant challenge, as large model sizes increase memory requirements and latency, limiting their use cases [60]. Quantization is a promising solution for reducing the LLM deployment costs [28, 69]. By lowering the model precision, quantization addresses both memory limitations and inference latency. The importance of quantization is pronounced for on-device deployments, where strict memory budgets often make model compression mandatory rather than optional. These scenarios typically require careful tuning of quantization levels to achieve an optimal balance between model size and quality [8, 14, 15, 58, 61].

While quantization may allow gigantic LLMs to fit into small-memory devices, it often leads to model quality degradation due to the inevitable loss of information. This is especially true for low-bit settings, such as 3-bit and 4-bit quantization, which are often used to accommodate the parameter sizes of LLMs [19,31,36]. This raises a key research question that this paper addresses: given a quantized LLM configured with the best possible effort under the memory budget, is there a way to recover the quality loss caused by quantization?

Leveraging external memory offers a potential solution to this problem. Specifically, on heterogeneous computing platforms where the CPU and GPU are connected via a PCIe interconnect—a common architecture in desktops and laptops—CPU memory becomes a viable option. Additional information that may be used to mitigate quantization errors can be stored in CPU memory and fetched at runtime, avoiding any additional GPU memory overhead.

However, utilizing CPU memory for GPU inference presents a critical challenge: the slow data transfer between the CPU and GPU can create a bottleneck for inference latency. To mitigate this issue, the volume of data transferred must be carefully controlled. Therefore, designing a system that effectively utilizes CPU memory requires the identification of the minimal set of additional information that can substantially enhance the quality of quantized models, while keeping the impact on latency minimal.

A clue for this problem comes from the well-known fact that not all channels in the weight matrix are equally important for quantization. Some channels, referred to as *salient channels*, are more critical, primarily due to the presence of activation outliers [6, 10, 25]. When certain input activation values are particularly large, the quantization errors in the corresponding weight channels—those multiplied by these large

*Equal contribution

activation values—are amplified, making the channels salient. By identifying these channels and selectively fetching error compensation terms from CPU memory, we can maximize the quality boost while minimizing data transfer.

Hence, the identification of the salient channels is essential for overcoming the bandwidth limitation, allowing for the transfer of only impactful information. Previous works that attempt to improve quantization algorithms by addressing activation outliers analyze the activation value distribution on a calibration data to predetermine the salient channels [10,27,33,36]. This approach is suboptimal as it statically designates certain channels as salient throughout the inference, while the real distribution of activation values—and thus the distribution of salient channels—changes dynamically at each decoding step. Accounting for this dynamic nature is crucial for the accurate identification of salient channels.

Thus, in this paper, we introduce DecDEC (**Dec**oding with **D**ynamic **E**rror **C**ompensation), an inference scheme for quantized LLMs that dynamically identifies salient channels and compensates for quantization errors in these channels, in real time. DecDEC enhances model quality while fully preserving the two main benefits of quantization: GPU memory savings and latency reduction. To achieve this, DecDEC stores the residuals of the quantized weight matrices in CPU memory, fetching only the parts that correspond to the dynamically identified salient channels for error compensation. Although small in size, these residuals provide a significant quality boost. This dynamic error compensation is performed concurrently with inference by an optimized GPU kernel, ensuring that all additional operations are seamlessly integrated into the existing workflow, minimizing inference slowdown. Below are the key contributions of our work:

- We provide an in-depth analysis on the dynamic nature of activation outliers in LLM inference.
- We present DecDEC, an inference scheme that enhances quantized LLMs by dynamically identifying salient channels and compensating quantization errors in them.
- We introduce a tuner for DecDEC that recommends system parameters to satisfy a target latency bound.
- We evaluate DecDEC across five different consumer-grade GPUs, demonstrating significant quality improvement with minimal memory and latency overhead.

2 Background

2.1 LLM Inference

Figure 1 presents a description of the modern LLM architecture and its inference flow. LLMs consist of multiple Transformer decoder blocks [59], each containing multiple linear layers, which account for most of the inference time, alongside other components like self-attention and normalization.

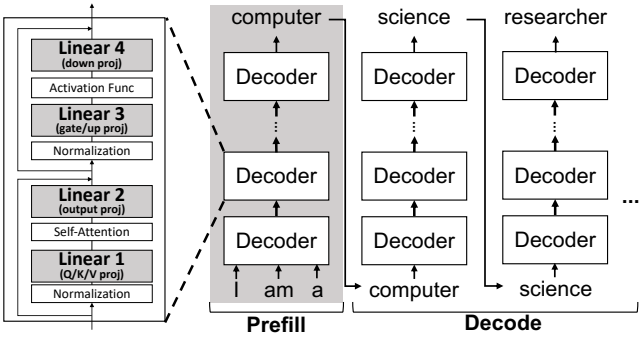


Figure 1: LLM inference.

LLM inference involves two phases: prefill and decode. In the prefill phase, all input tokens (e.g., 'I', 'am', 'a') are processed in parallel to generate a single output token (e.g., 'computer'). The decode phase begins subsequently, where the output token of the previous step is fed back into the model to generate the next token, repeated until the end of the sequence. This sequential nature of the decode phase makes it the primary latency bottleneck.

The decode phase is particularly memory-bound, as only one token is processed at a time, reducing the linear layers to GEMV operations. In data center settings, this issue can be alleviated by batching together multiple queries [32,66]. However, this usually cannot be applied to on-device inference, where LLMs serve only individual users.

2.2 LLM Quantization

Quantization for LLMs—a popular compression technique that reduces both memory usage and inference latency—can be categorized into two main types [28,69]: weight-activation quantization [10, 34, 37, 49, 64, 65] and weight-only quantization [9, 11, 12, 17, 19, 30, 31, 33, 36, 57]. Each are suited to different inference scenarios. Weight-activation quantization is primarily used in datacenter settings, where both memory and computational costs must be minimized to improve throughput. Quantizing both weights and activations allows for the efficient use of low-precision arithmetic units (e.g., INT4, INT8, FP8) available on modern GPUs [1]. In contrast, for on-device inference, weight-only quantization is the preferred approach [31, 69]. In this apporach, quantized weights are loaded from memory and dequantized on-the-fly to full precision (i.e., FP16), before being multiplied with the full-precision activations [23]. Though it only reduces memory traffic, this is sufficient to speed up on-device inference where memory is the bottleneck, as discussed in Section 2.1.

Weight-only quantization can be further divided into two sub-categories: quantization-aware training (QAT) [11,30,37], and post-training quantization (PTQ) [9, 12, 17, 19, 31, 33, 36, 57]. While QAT yields better results by retraining to reduce quantization errors, its cost makes it impractical for many end-

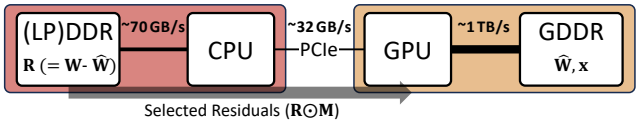


Figure 2: CPU-augmented inference for quantized LLMs.

users [31,69]. As a result, PTQ—requiring no retraining—has become the preferred method for on-device LLM inference. Therefore, in this paper, we specifically focus on weight-only PTQ for on-device inference.

3 Augmenting Quantized LLM with CPU Memory

In this section, we propose leveraging CPU memory—an often underutilized resource in LLM inference, as GPUs are the de facto standard processors for this workload—as a means to augment quantized LLMs. Section 3.1 introduces the concept of CPU-augmented quantized LLMs while Section 3.2 and Section 3.3 explore its opportunities and challenges.

3.1 Concept

Goal. We aim to leverage CPU memory to improve quantized LLM quality without additional GPU memory costs. Quantization trades model quality for a smaller memory footprint. In constrained settings, practitioners optimize this trade-off within a fixed GPU memory budget by selecting a uniform bitwidth or applying fine-grained strategies such as layer-wise [8,14,15] or channel-wise [58,61] allocation. Our goal is to then further improve quality post hoc by utilizing CPU memory, without increasing GPU memory usage.

Basic Mechanism. Figure 2 illustrates our concept of leveraging CPU memory to enhance quantized LLMs. We primarily target desktop or laptop platforms, where the GPU is connected to the CPU via a PCIe interconnect. As in conventional inference systems, the quantized weight parameters ($\widehat{\mathbf{W}}$) and activations ($\mathbf{x}$) are kept in GPU memory. The difference here is that $\mathbf{R}$—the residual between the original full-precision weights and the quantized weights—is stored in CPU memory. During the decode phase, residuals are fetched from the CPU to help compensate for quantization errors, potentially improving model quality. Due to the limited bandwidth of PCIe, which is typically an order of magnitude lower than GPU memory bandwidth (e.g., 32 GB/s vs. 1 TB/s), fetching the entire residual matrix would incur a prohibitive latency bottleneck. Therefore, only a small subset of residuals should be fetched in a selective manner. In short, this process augments each linear operation of quantized LLM from $\widehat{\mathbf{W}}\mathbf{x}$ to $(\widehat{\mathbf{W}}+\mathbf{R}\odot\mathbf{M})\mathbf{x}$, where $\mathbf{M}$ is a binary mask that sparsifies $\mathbf{R}$.

Key Research Question. A key research question in designing an effective CPU-augmented inference system for quantized LLMs is determining a subset of residuals, or mask $\mathbf{M}$. A good mask $\mathbf{M}$ should: 1) select portions of the residuals that contribute most to improving model quality within the bandwidth constraints, and 2) maintain a structured form that minimizes indexing overhead and ensures effective residual transfer, while enabling efficient processing on the GPU.

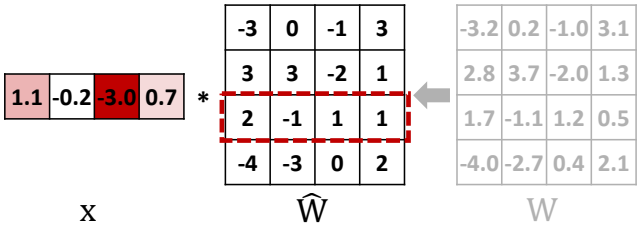


Figure 3: Activation outlier issue in weight quantization.

3.2 Opportunity: Not All Residuals Are Equally Important

Some residuals are more important than others—an opportunity that can be leveraged to determine $\mathbf{M}$. This opportunity arises from the presence of activation outliers (i.e., activation values with large magnitudes), a well-known phenomenon in LLM inference [6,10,25,27,33,36]. When certain activation values are noticeably large, even small quantization errors in the corresponding weight channels can be multiplied and amplified, leading to considerable perturbations in the output. Figure 3 illustrates this issue. In this example, the third input channel (third row) of the weight matrix is multiplied by the activation outlier, -3.0. We refer to such channels as *salient* channels. Constructing $\mathbf{M}$ at the input channel granularity based on the magnitude of input activations can satisfy two key conditions for an effective mask: selecting impactful portions of the residuals and maintaining a structured form.

Indeed, compensating for errors in salient channels using the corresponding residuals is highly effective. Figure 4 illustrates how the quantization error, defined as the mean squared error between the computation result with FP16 weights ($\mathbf{Wx}$) and quantized weights ($\widehat{\mathbf{W}}\mathbf{x}$), is reduced by sequentially replacing the input channels of the quantized weight with their corresponding FP16 values. For this analysis, we evaluate 3-bit and 4-bit versions of the LLaMA-3-8B-Instruct model [16], quantized using the state-of-the-art method AWQ [36], with a text sample from the C4 dataset [47] as the input prompt. All four linear layers in the 8th, 16th, and 24th decoder blocks are included in this evaluation. The quantization errors for both the 3-bit and 4-bit models drop rapidly when we progressively compensate for channels in descending order of their activation magnitudes (solid red and solid blue lines). This trend closely follows the activation magnitude distribution (solid black line), which represents the sorted activation magnitudes in descending order. In contrast, the reduction in

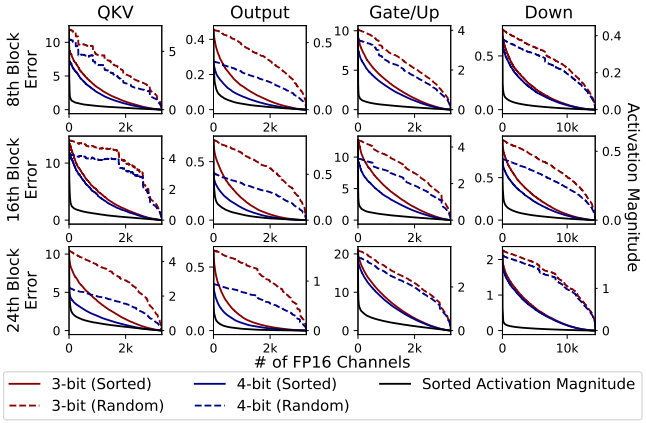


Figure 4: Quantization error reduction trends observed when replacing the input channels of quantized weights with FP16 values sequentially in sorted order (solid lines) and random order (dotted lines) for the 8th, 16th and 24th decoder block of Llama-3-8B-Instruct. The distribution of activation magnitudes in sorted order is also shown (black lines).

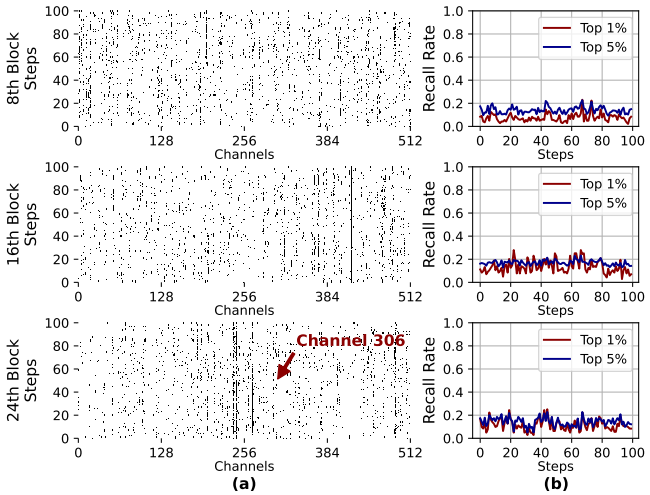


Figure 5: (a) Distribution of activation outliers (top 5%) and (b) Recall rate of static analysis-based outlier identification for the true top 1% and 5% outliers, across 100 decoding steps. The down projection layers in the 8th, 16th, and 24th decoder blocks of Llama-3-8B-Instruct model are used for profiling.

quantization error is significantly slower when input channels are compensated in random order, as shown by the dotted lines. This highlights the importance of prioritizing salient channels based on activation magnitude.

3.3 Challenge: Dynamic Nature of Activation Outliers

Identifying salient channels is challenging because the distribution of activation outliers changes dynamically by nature. While it is possible to infer salient channels by statically analyzing activation value statistics on a small calibration set [10, 27, 33, 36], such static approaches are suboptimal. Figure 5(a) shows the distribution of activation outliers—defined as activations with the top 5% magnitudes—in the down projection layer of the 8th, 16th, and 24th decoder blocks of the Llama-3-8B-Instruct model over 100 decoding steps, using a text sample for C4 dataset as the input prompt [47]. For visibility, only the first 512 channels are shown. While some channels (e.g., Channel 306 in 24th block, highlighted by an arrow) consistently exhibit high activation magnitudes and remain persistent outliers, the outlier distribution generally shows significant irregularity across decoding steps.

To quantify the dynamic nature of activation outliers, we calculate the recall rate of the top 1% and top 5% outliers identified through static analysis using a calibration set, compared to the true top 1% and top 5% outliers (ground truth) observed at each decoding step. We use a subset of the Pile dataset [21] for calibration, following prior work [36]. Specifically, we profile the average of the mean square of each activation value and use this as a metric for identifying outliers. Figure 5(b) presents the results, showing that the recall rate remains low (∼20%) for both the top 1% and top 5% outliers. This highlights a clear limitation of static analysis, as it misses the majority of outliers at runtime, emphasizing the need for dynamic identification of salient channels.

4 DecDEC Design

4.1 Overview

In this section, building on the opportunity outlined in Section 3.2 and addressing the challenge described in Section 3.3, we propose DecDEC, a CPU-augmented inference system for quantized LLMs that performs decoding with dynamic error compensation. Figure 6 presents an overview of DecDEC. During the decode phase, DecDEC augments each linear layer, essentially a GEMV operation, with dynamic error compensation. To produce the final output $\mathbf{o}$, DecDEC adds an error compensation term $\mathbf{o}_{\text{dec}}$ to the base GEMV result $\mathbf{o}_{\text{b}} = \widehat{\mathbf{W}}\mathbf{x}$. $\mathbf{o}_{\text{dec}}$ is computed by multiplying the input vector with a subset of weight residuals selectively fetched from the CPU. This selection is performed *dynamically*, fully accounting for the variability of input vectors. To maximize the number of residual values fetched under PCIe bandwidth constraints, DecDEC stores and retrieves a quantized version of the residuals ($\widehat{\mathbf{R}}$), comprising the quantized values $Q_r(\mathbf{R})$ and associated metadata, instead of the full-precision residuals ($\mathbf{R}$). Here, Q_r is a quantizer that maps full precision residuals to low-bit form and differs from the base quantizer used for the weights, Q_b.

The dynamic error compensation process consists of four sequential steps. ❶ First, by investigating the input activation vector, DecDEC creates `sc_indices`, a list of salient channel

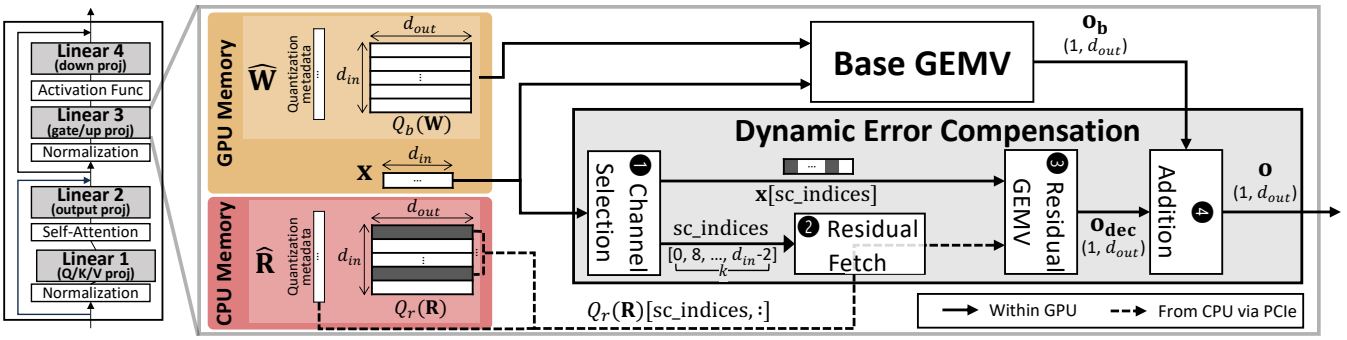


Figure 6: DecDEC overview.

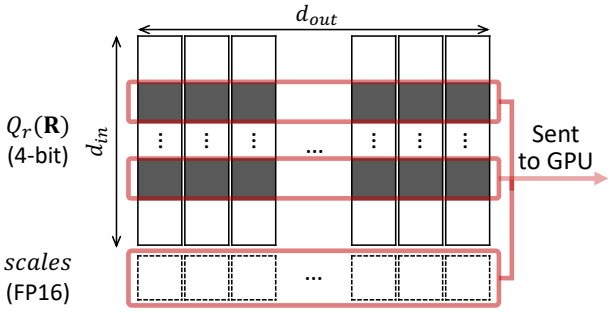


Figure 7: Quantization of weight residual.

indices. The number of salient channels to compensate, k, is a preconfigured parameter. This step is essentially a Top-K operation that selects the values in the input activation vector with the largest magnitudes. ❷ Next, a portion of the quantized residuals corresponding to the salient channels, $Q_r(\mathbf{R})[\texttt{sc_indices},:]$ (along with the necessary quantization metadata), is fetched from the CPU via PCIe. ❸ The fetched residuals are then multiplied by the sparsified activation vector ($\mathbf{x}[\texttt{sc_indices}]$), producing $\mathbf{o_{dec}}$. ❹ Finally, the resultant $\mathbf{o_{dec}}$ is added to the base GEMV result $\mathbf{o_b}$, producing the final output, $\mathbf{o}$. All steps run in parallel with the base GEMV on a different GPU stream, and must be highly efficient to remain hidden within base GEMV runtime.

The following sections provide details of each component of DecDEC. Section 4.2 explains how DecDEC performs the residual quantization. Section 4.3 details the GPU implementation of the dynamic error compensation. Section 4.4 illustrates how DecDEC configures system parameters, including k, the number of channels to compensate.

4.2 Residual Quantization

Figure 7 depicts the quantization scheme for the residuals. DecDEC employs 4-bit quantization for each output channel (i.e., column) of the residuals. To minimize metadata, symmetric uniform quantization is used. This approach requires only a single scalar scale factor as metadata for each output channel. The residual quantizer for the i-th output channel ($Q_{r,i}$) is defined as:

$$Q_{r,i}(r) = \text{clip}\left(\text{round_to_int}\left(\frac{r}{S_i}\right), -7, 7\right).$$

S_i, the scale factor, is determined through a grid search as the value that minimizes the mean squared error between the original and quantized weights.

Using this quantizer, each floating-point residual value r is projected to an integer between -7 and 7. At runtime, the selected input channels of the quantized residuals (highlighted in Figure 7) and all the scale factors are fetched from the CPU. Each input channel of the quantized residuals, as well as the scales, are stored contiguously in CPU memory, enabling coalesced data transfers.

4.3 Efficient Implementation of Dynamic Error Compensation

The top priority in implementing dynamic error compensation is ensuring low latency, allowing its execution to remain hidden within the base GEMV execution time. To fetch a sufficient number of residual channels within this short time window, DecDEC introduces three key software optimization strategies: 1) zero-copy residual fetching, 2) fast approximate Top-K for channel selection, and 3) kernel fusion.

Zero-Copy Residual Fetch. DecDEC leverages CUDA zero-copy [42], instead of commonly used API functions such as `cudaMemcpy()` or `cudaMemcpyAsync()`, to fetch the residuals from CPU. These APIs rely on the direct memory access (DMA) engine for data transmission, which is efficient at transferring large data blocks but suboptimal for smaller transfers due to the DMA setup overheads. Fetching residuals, however, falls into the latter category. The granularity of residual fetching occurs at the row level of the quantized residual matrix. With 4-bit quantization and typical row lengths of a few thousand to tens of thousands, each data block transfer is only a few tens of KBs. For optimal PCIe bandwidth utilization, the data block size should ideally be at least a few

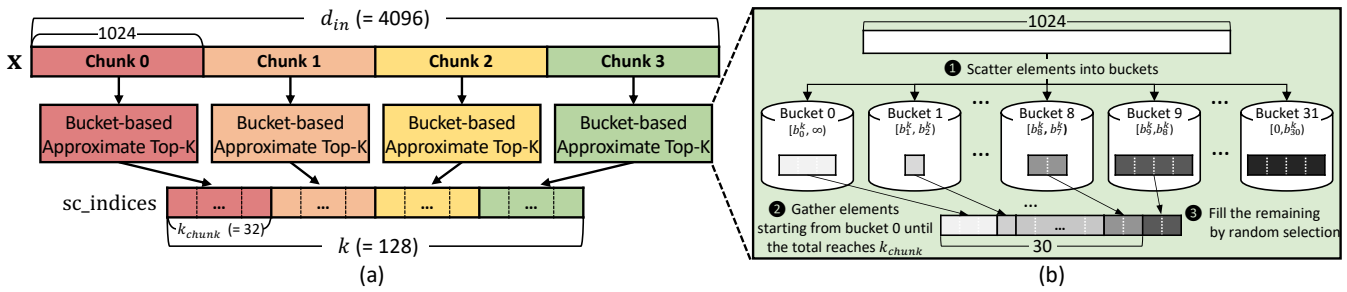


Figure 8: Fast approximate Top-K operation of DecDEC.

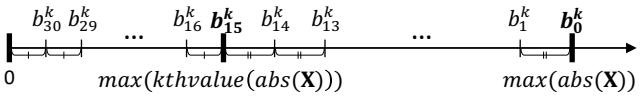


Figure 9: DecDEC 's approximate Top-K bucket boundaries.

hundred KBs (e.g., 256 KB) [41, 46]. In zero-copy access, the GPU directly sends cacheline-sized memory requests, making it suitable for fine-grained data access. While zero-copy access has the disadvantage of occupying GPU cores to generate memory requests—potentially slowing down other concurrently running kernels—this is not a major issue for DecDEC. The concurrent kernel for our case, the base GEMV, is typically memory-bound, so using fewer cores for this kernel is unlikely to have significant impact on its execution time.

Fast Approximate Top-K for Channel Selection. For channel selection, instead of an exact Top-K operation, DecDEC employs an approximate Top-K method that is fast and GPU-friendly while maintaining precision. Figure 8 illustrates the approach using an example where 128 elements are selected from a 4096-dimensional activation vector ($d_{in} = 4096$, $k = 128$). As shown in Figure 8(a), instead of a single global selection, DecDEC partitions the input into four contiguous 1024-dimensional chunks and performs a local Top-k_{chunk} selection within each chunk. In this example, $k_{chunk} = k/4 = 32$. The locally selected elements are then concatenated to form the final result. Although this introduces approximation, it significantly reduces latency by avoiding global synchronization—each local selection is handled independently by a thread block. A larger chunk size can improve efficiency but may increase the approximation error; we set the chunk size to 1024 to balance this trade-off effectively.

For each local selection, DecDEC uses a variant of the bucket-based Top-K algorithm [2]. Figure 8(b) illustrates this process. ❶ First, the 1024 elements in a chunk are scattered into buckets based on their magnitudes, with bucket boundaries (b_0^k, b_1^k, ..., b_{30}^k). The number of buckets is set to 32, matching the number of threads in a warp, allowing for efficient thread-level parallelism so that each thread processes a different bucket. ❷ Next, elements are gathered, starting from bucket 0, until the total count reaches k_{chunk}. ❸ If the number of elements in the current bucket exceeds the remaining spots for k_{chunk}, as in the case of bucket 9 in Figure 8(b), random selection is used to fill the remaining spots. This random selection adds an additional layer of approximation but significantly reduces latency by avoiding exact sorting.

Determining proper bucket boundaries is crucial to minimize the approximation error introduced by random selection (❸ in Figure 8(b)). DecDEC profiles the distribution of activation values using a small calibration set and aims to set boundary values that balance Top-K accuracy with the ability to handle a broader range of values. Placing finer-grained buckets around the expected k-th largest value generally improves accuracy, but limits the system's ability to handle out-of-distribution values. To address this, DecDEC uses an offline analysis to determine two key boundaries, b_0^k and b_{15}^k, from which the other boundaries are inferred (as shown in Figure 9). Let the distribution of activation values from the calibration set be $\mathbf{X} \in R^{N \times d_{in}}$, where N is the size of the set. The boundary b_{15}^k is set to the maximum of the k-th largest value across all vectors in $abs(\mathbf{X})$. The interval between 0 and b_{15}^k is uniformly divided into 16 buckets (b_{15}^k, b_{16}^k, ..., b_{30}^k), focusing on the range where the k-th largest value is most likely to occur. To handle out-of-distribution cases which can cause significant degradation in selection precision, DecDEC assigns an additional 16 buckets for values beyond b_{15}^k. Specifically, b_0^k is set to the maximum value in all $abs(\mathbf{X})$, and the range between b_0^k and b_{15}^k is also uniformly divided to form the remaining 16 buckets.

Kernel Fusion. DecDEC extensively fuses all dynamic error compensation operations into a single kernel. Figure 10 visualizes the execution flow of the fused kernel, using an example where two thread blocks process a weight matrix of size 4096×6144. In this example, only one channel is selected per each of the four chunks (i.e., $k_{chunk} = 1$, $k = 4$). Initially, each thread block sequentially processes two chunks, selecting a total of two channels (Step ❶ in Figure 6(b)). The indices for the selected channels (`sc_indices`) as well as the corresponding activation values ($\mathbf{x}$[`sc_indices`]) are stored in GPU memory. Thread blocks are then synchronized using the grid-wide synchronization feature of the cooperative group [43]. This synchronization is required because each thread block

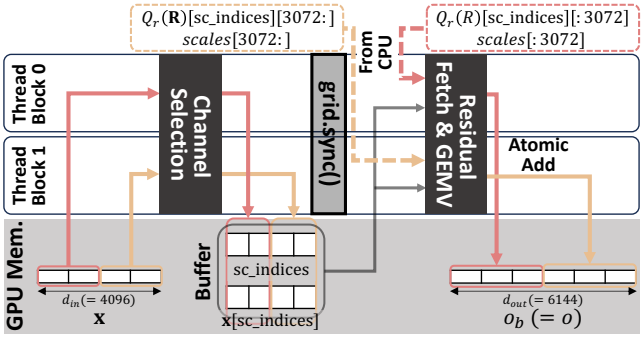


Figure 10: Fused kernel for dynamic error compensation.

processes a segment of all selected channels—not just a disjoint subset—when fetching quantized residuals from the CPU and performing GEMV (Steps ❷ and ❸ in Figure 6(b)). For example, thread block 0 processes $Q_r(\mathbf{R})[\texttt{sc_indices}][:3072]$ as opposed to $Q_r(\mathbf{R})[\texttt{sc_indices}[:2]][:]$. The thread block-level synchronization allows all thread blocks to have access to the complete set of `sc_indices` and $\mathbf{x}[\texttt{sc_indices}]$. This partitioning scheme allows for efficient reduction in the residual GEMV without requiring extensive global synchronization. The results from the residual GEMV are directly added to the result of the base GEMV ($\mathbf{o_b}$) using atomic primitives, yielding the final output ($\mathbf{o}$) (Step ❹ in Figure 6(b)).

GPU Memory Overhead. The buffer for `sc_indices` and $\mathbf{x}[\texttt{sc_indices}]$ in the fused kernel is the only additional GPU memory usage of DecDEC. Zero-copy from the CPU does not consume GPU memory. The bucket boundary values are not stored on the GPU; instead, only b_0^k and b_{15}^k are passed to the kernel as arguments. A single buffer can be reused for all linear layers if it is sized in accordance with the largest k. In the extreme case of fetching 10% of the channels across all layers in Llama-3-8B, the maximum k would be 1433, for the down projection layer. This calls for an 8.6 KB buffer ($1433 \times (4+2)$), which is less than 0.0003% of the model size, assuming 3-bit precision—essentially a negligible overhead.

4.4 Parameter Tuner

Necessity of Parameter Tuner. Effective use of DecDEC requires careful tuning two key parameters:

- n_{tb}: Specifies the number of thread blocks used for dynamic error compensation. Since DecDEC runs in parallel with the base GEMV, allocating too many thread blocks to dynamic error compensation can slow down the base GEMV computation, while allocating too few may underutilize PCIe bandwidth, as zero-copy transfers require GPU cores to issue memory requests.
- k_{chunk}: Specifies how many channels are compensated per chunk (1024 channels). A larger k_{chunk} improves model quality but may increase inference latency.

Choosing n_{tb} and k_{chunk} is challenging due to the large design space. The n_{tb} value for each type of linear layer (n_{tb}^{qkv}, n_{tb}^{o}, n_{tb}^{gu}, n_{tb}^{d}) has multiple viable candidates, constrained by the dimensions of the corresponding weight matrices. For example, in Llama-3-8B, there are 9 possible candidates for n_{tb}^{qkv} (1, 2, 3, 4, 5, 6, 8, 12, 24), while other layers also have multiple options. Selecting k_{chunk} presents a greater challenge due to its broader range of possibilities. The k_{chunk} value for each type of linear layer (k_{chunk}^{qkv}, k_{chunk}^{o}, k_{chunk}^{gu}, k_{chunk}^{d}) can be any integer less than a maximum determined by the available shared memory. Although platform-dependent, this upper bound is large (e.g., 367 with 48 KB per-block shared memory), leading to an expansive search space. This creates a combinatorial explosion of configuration options. Technical details on how to identify candidate values of k_{chunk} and n_{tb} for a given model and platform are presented at the end of this section.

To address this issue, we provide a DecDEC tuner that suggests n_{tb} and k_{chunk} values based on a target slowdown rate. The tuner maximizes k_{chunk} while keeping the total execution time—including base GEMV and dynamic error compensation across all linear layers—within the specified slowdown relative to the baseline (i.e., without compensation). This tuning is a one-time process for a given model-device pair.

Parameter Tuning Process. Figure 11(a) illustrates the tuning process, which consists of two phases: Phase 1 determines the n_{tb} values, and Phase 2 determines the k_{chunk} values.

In Phase 1, the tuner simplifies the search for n_{tb} values for each layer by replacing it with the search for a single metaparameter, n_{tb}^{max}. n_{tb}^{max} represents the upper limit on the number of thread blocks to use for dynamic error compensation. Each layer's n_{tb} is then set to the largest candidate below n_{tb}^{max}. Determining n_{tb}^{max} involves testing values up to half of the total SM count, to reduce the search space. In the example shown in Figure 11, the GPU has 56 SMs, so values up to 28 are tested. For each tested n_{tb}^{max}, a coarse-grained k_{chunk} search checks how many uniform increments to k_{chunk} can be applied across all layers without exceeding the target slowdown rate. Figure 11(b) shows an example of a coarse-grained k_{chunk} search for $n_{tb}^{max} = 24$ (i.e., $(n_{tb}^{qkv}, n_{tb}^{o}, n_{tb}^{gu}, n_{tb}^{d}) = (24, 16, 23, 14)$), yielding 19 valid steps. If no steps can be made for any n_{tb}^{max} value, the tuner fixes k_{chunk} to 0 for the layer with the smallest weight matrix and repeats the process, as smaller matrices are often most sensitive to increases in k_{chunk}.

After Phase 1, the tuner selects the n_{tb}^{max} with the most steps and proceeds to a fine-grained k_{chunk} search in Phase 2. Figure 11(c) shows the fine-grained k search for the selected $n_{tb}^{max} = 24$. In this phase, not all k_{chunk} values may increase together. At each step, the tuner increments k_{chunk} for as many layers as possible, prioritizing those with smaller increases in execution time. For example, in Step 1, k_{chunk}^{gu} is incremented first, followed by k_{chunk}^{d}. Step 1 stops at this point, as further increases to k_{chunk}^{qkv} and k_{chunk}^{o} would exceed the target; thus

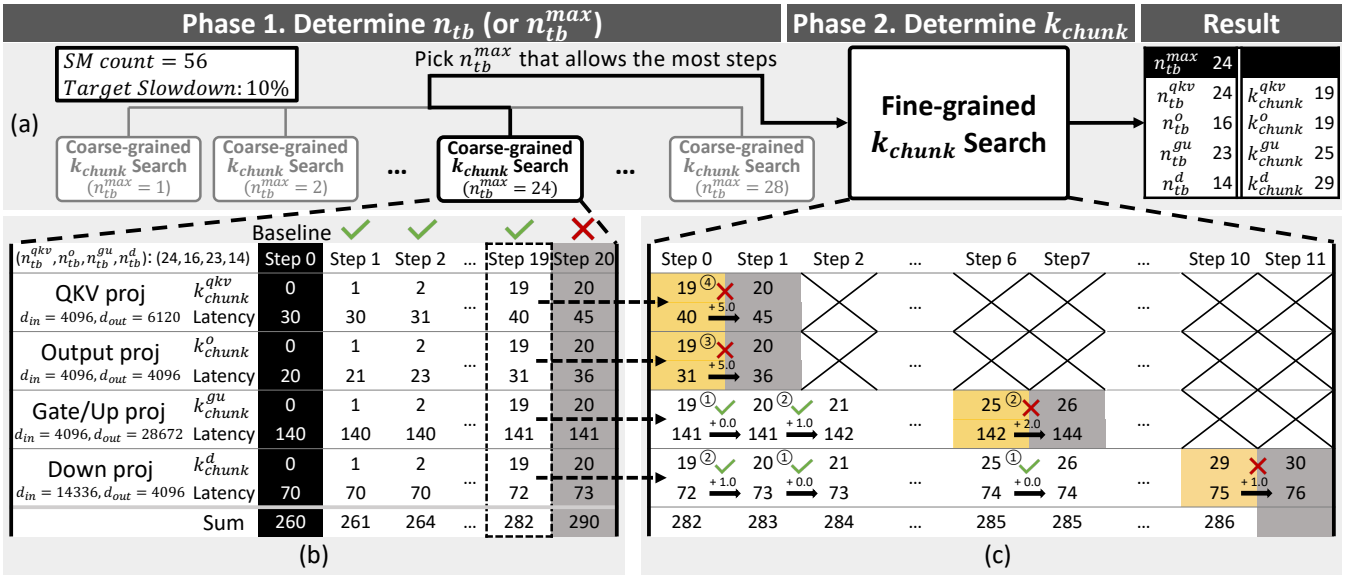


Figure 11: Parameter tuning process for DecDEC, assuming a total of 56 SMs and a target slowdown rate of 10%.

their final values are set (i.e., $k_{chunk}^{qkv}, k_{chunk}^{o} = 19$). This process repeats until no further increments can be made for any layer.

Technical Details: n_{tb} and k_{chunk} Candidates. The n_{tb} values considered during parameter tuning are those that have a meaningful impact on at least one of the two parts of the kernel execution: the approximate Top-K selection and the residual fetching. In the approximate Top-K selection part, the minimum processing granularity per thread block is one chunk. Therefore, increasing n_{tb} beyond the number of chunks does not have additional effect on performance. Thus, the set of n_{tb} values relevant to this part is:

$$A = \left\{ n \;\middle|\; 1 \le n \le \frac{d_{in}}{1024} \right\}.$$

For residual fetching part, 4-bit residuals are transferred over PCIe in coalesced segments of 256 values (128 bytes), resulting in a total of $s = d_{out}/256$ segments. These segments are distributed across n_{tb} thread blocks, with each block processing $\lceil s/n_{tb} \rceil$ segments. If multiple n_{tb} values result in the same number of segments per block ($\lceil s/n_{tb} \rceil$), only the smallest such value is considered and the rest are redundant. Excluding these cases, the candidate set relevant to this part is:

$$B = \left\{ n \;\middle|\; 1 \le n \le s, \left\lceil \frac{s}{\lceil s/n \rceil} \right\rceil = n \right\}.$$

The final candidate set for n_{tb} is the union of the two (i.e., $N = A \cup B$).

Meanwhile, k_{chunk} is bounded by the shared memory limit. During the approximate Top-K selection part of the kernel, shared memory usage increases with the value of k_{chunk}. Specifically, the shared memory usage for this part is:

$$128 + 128 \times k_{chunk} + 2 \times 1024 \text{ bytes}.$$

GPU Name	Memory Size	Memory BW	# SM	PCIe BW	R_{bw}
		Desktop			
RTX 4090	24 GB	1,008 GB/s	128	32 GB/s	32
RTX 4080S	16 GB	736 GB/s	80	32 GB/s	23
RTX 4070S	12 GB	504 GB/s	56	32 GB/s	16
		Laptop			
RTX 4070M	8 GB	256 GB/s	36	16 GB/s	16
RTX 4050M	6 GB	192 GB/s	20	16 GB/s	12

Table 1: GPU specifications.

Here, 128 bytes are used for integer counters that track the number of elements that fall into each of the 32 buckets; $128 \times k_{chunk}$ bytes are used to temporarily store the indices of elements assigned to each bucket; and 2×1024 bytes account for the input activation values in the chunk. The total must remain below the per-block shared memory limit (e.g., 49,152 bytes), which constrains the maximum allowable k_{chunk} value.

5 Evaluation

5.1 GPU Kernel Benchmarks

Methodology. In this section, we evaluate the DecDEC GPU kernel on three consumer-grade GPUs: two desktop GPUs, RTX 4070 Super (RTX 4070S) and RTX 4090, and one laptop GPU, RTX 4050 Mobile (RTX 4050M), with their specifications detailed in Table 1. Here, R_{bw} denotes the ratio of memory bandwidth to CPU-to-GPU (PCIe) bandwidth. The evaluation considers GEMV operations of output, gate/up and down projection layers in Llama-3-8B-Instruct [16], assuming 3-bit bitwidth. For the base GEMV, we use the LUTGEMM kernel, [23], a state-of-the-art GEMV kernel for uniform quantization. Kernel times are measured using NVIDIA Nsight Systems.

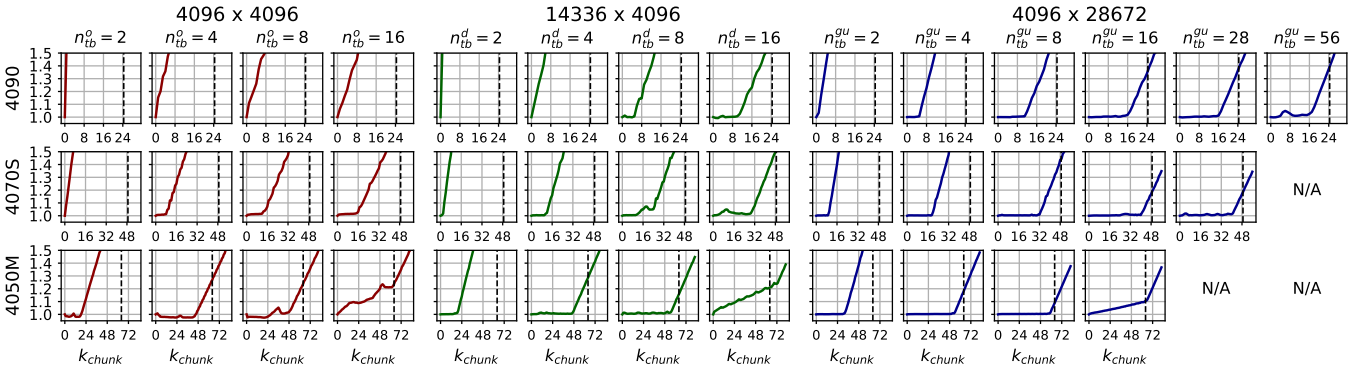


Figure 12: Execution time of base GEMV + DecDEC with varying k_{chunk} and n_{tb}, normalized to base GEMV execution time.

Expected Behavior. The execution time of the DecDEC kernel is expected to follow a piecewise linear function of k_{chunk} with two distinct segments. In the first segment, corresponindg to small k_{chunk} values, the latency remains nearly constant at the base GEMV time, as the compensation operations are fully hidden under the GEMV execution. In the second segment, once k_{chunk} exceeds a certain threshold—the knee point—the execution time linearly increases with k_{chunk}, inidcating that the compensation operations are no longer fully hidden. Theoretically, the knee point is expected to occur when $k_{chunk} = 1024 \times {}^{1}/_{R_{bw}} \times {}^{3}/_{4}$, representing the maximum amount of data transfer that can overlap with the base GEMV. Here, the base GEMV time is approximated by dividing the weight matrix size by the GPU memory bandwidth, and PCIe bandwidth is assumed to be fully utilized. The factor ${}^{3}/_{4}$ accounts for the 3-bit quantization; for other bitwidths, this factor should be adjusted accordingly (e.g., ${}^{4}/_{4} = 1$ for 4-bit quantization).

Results. Figure 12 shows the execution time of DecDEC kernel (base GEMV + dynamic error compensation), normalized to the standalone execution time of the base GEMV, across varying n_{tb} and k_{chunk}. Each subfigure also includes a vertical dotted line to mark the theoretical knee point derived from the analytical model (i.e., $1024 \times {}^{1}/_{R_{bw}} \times {}^{3}/_{4}$). All cases exhibit the expected two-segment piecewise linear behavior when n_{tb} is properly set, except for the 4096×4096 case on RTX 4090, the fastest GPU evaluated. In this case, the base GEMV execution time is so short that even a small k_{chunk} incurs overhead.

From the other cases, three key observations can be made. First, a lower R_{bw}—indicating lower memory bandwidth and higher PCIe bandwidth—shifts the knee point to the right. Thus, RTX 4050M, with the lowest R_{bw}, supports the largest k_{chunk} before the knee point, while RTX 4090, with the highest ratio, supports the smallest. This trend is consistent with the ordering predicted by theoretical knee points. Second, the knee point is highly sensitive to n_{tb}, highlighting the importance of careful tuning. Generally, higher n_{tb} values such as 8 or 16 delay the knee point. In contrast, small n_{tb} values (e.g., 2) cause the knee point to appear too early or disappear, leading to suboptimal performance. However, increasing n_{tb} can sometimes worsen results by slowing down the base GEMV, as discussed in Section 4.4. This effect is particularly noticeable on GPUs with fewer SMs, such as the 4050M. For instance, on the 4050M, $n_{tb} = 8$ yields the best results, whereas increasing n_{tb} to 16 leads to worse performance. Third, larger weight sizes enable higher k_{chunk} with minimal latency overhead, due to increased time slack. With sufficiently large weight sizes (e.g., 4096×28672) and a properly tuned n_{tb}, the actual knee point approaches the theoretical value. For example, on an RTX 4050M with a 4096×28672 matrix and $n_{tb} = 8$, the observed knee point is around $k_{chunk} = 60$, compared to a theoretical prediction of 64.

5.2 Impact on Model Quality

Methodology. We demonstrate DecDEC 's quality improvements on 3-bit, 3.5-bit, and 4-bit versions of two instruction-tuned LLMs: Llama-3-8B-Instruct [16] (hereafter referre to as Llama-3) and Phi-3-medium-4k-instruct (14B model, hereafter referred to as Phi-3) [40]. For the 3.5-bit version, we adopt a block-wise bitwidth allocation, applying 3-bit quantization to half of the decoder blocks and 4-bit quantization to the remaining blocks. This follows a KL divergence-based sensitivity metric from prior work [8].

As base quantization methods, we choose two state-of-the-art LLM quantization methods: AWQ [36] and SqueezeLLM [31]. AWQ is a *uniform* quantization method that mitigates quantization errors by applying per-channel scaling to protect salient channels, which are identified through an offline analysis on a calibration dataset. SqueezeLLM employs a clustering-based *non-uniform* quantization method that considers the sensitivity of each weight.

We evaluate DecDEC-augmented models across varying k_{chunk}, the number of channels compensated per chunk (1024 channels). We uniformly set k_{chunk} to 8, 16, 32, 64 and 128 for all layers. Based on the results in Section 5.1, values up to 64 fall within the practical range—where latency overhead from

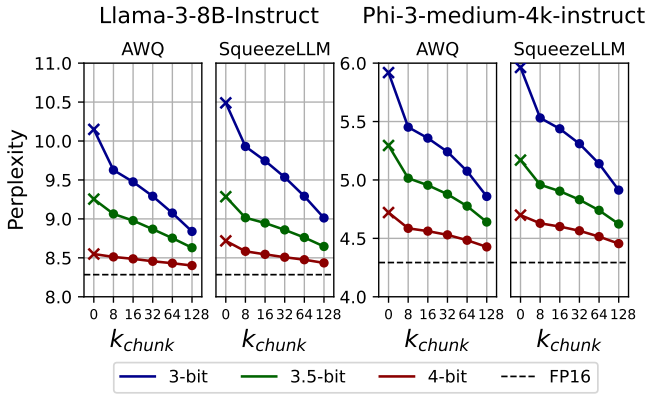


Figure 13: Perplexity on WikiText. The x markers correspond to baselines without DecDEC ($k_{chunk} = 0$). Lower is better.

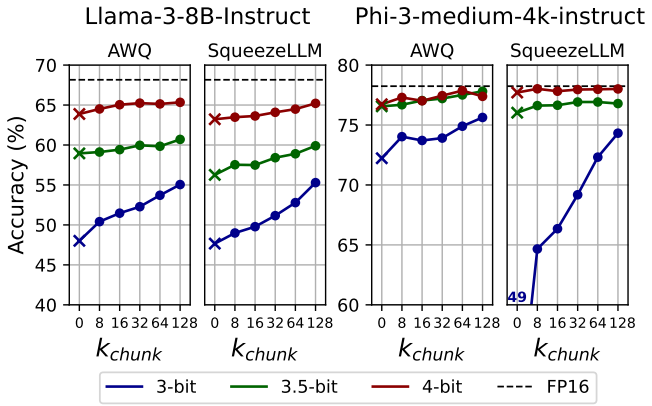


Figure 14: Accuracy on BBH. The x markers correspond to baselines without DecDEC ($k_{chunk} = 0$). Higher is better.

dynamic error compensation may remain low depending on the platforms—while $k_{\text{chunk}} = 128$ is included for completeness in evaluating upper-bound behavior.

Benchmarks. Following previous literature [9, 12, 31, 36], we use perplexity on WikiText [39] as the primary metric, as it reliably reflects quantized LLM quality [13, 36]. Additionally, we use BIG-Bench Hard (BBH) [54], a collection of 23 challenging tasks in BIG-Bench [52], to assess the models' capabilities in problem solving. Chain-of-Thought (CoT) is enabled for all BBH evaluations [62]. Lastly, we evaluate multi-turn conversation using MT-Bench [68], where a strong LLM judges 80 responses, scoring each from 0 to 10. We use GPT-4o as the judge and report the average of three runs.

Perplexity on WikiText. Figure 13 shows the perplexity results. Across all cases, a clear trend is observed: perplexity consistently decreases (i.e., model quality improves) as k_{chunk} increases. In particular, for 3-bit models, significant improvements occur even at $k_{chunk} = 8$. AWQ's perplexity decreases from 10.15 to 9.63 for Llama-3 and from 5.96 to 5.53 for Phi-3, while SqueezeLLM's perplexity drops from 10.49 to

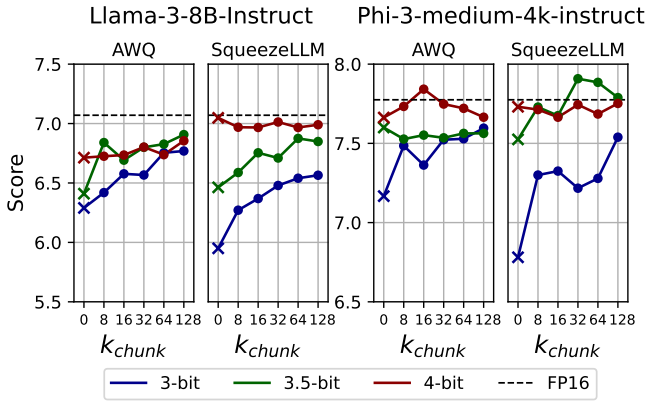


Figure 15: MT-Bench scores. The x markers correspond to baselines without DecDEC ($k_{chunk} = 0$). Higher is better.

9.93 for Llama-3 and from 5.92 to 5.45 for Phi-3. Meanwhile, the impact on 4-bit models is relatively less pronounced. This is expected, as 4-bit models are already close to full-precision, leaving less room for improvement. The 3.5-bit models follow an intermediate trend.

BBH and MT-Bench. Results on BBH (Figure 14) follow the same trends as the perplexity results. The MT-Bench results (Figure 15), on the other hand, demonstrate some different patterns. In cases where vanilla quantized models without DecDEC ($k = 0$) already achieve scores very close to those of the FP16 model—such as all 4-bit cases and the AWQ 3.5-bit model of Phi-3—the scores remain unchanged, oscillating around the baseline score. For the remaining cases, DecDEC significantly improves scores even with a small k_{chunk} (e.g., 8), like in other benchmarks; further increases in k_{chunk}, however, do not always yield noticeable improvements. These patterns may be attributed to the coarse-grained rubric of this benchmark, which assigns integer scores ranging from 0 to 10 for each task. This coarse-grained scoring may miss subtle improvements when the potential gain is small.

Effectiveness of DecDEC's Channel Selection. Figure 16 compares DecDEC with three variants using different channel selection mechanisms. To assess the benefits of dynamic selection, we include *Random* (blue line), which selects channels randomly, and *Static* (yellow line), which statically select channels by Hessian-based ranking with a calibration set (using exact sorting) following prior work [33]. To isolate the effect of DecDEC's Top-K approximation, we include *Exact* (red line), which uses true Top-K channels. Alongside perplexity results, we also report the average recall rate of Static and DecDEC relative to Exact.

While Static improves over Random, it underperforms DecDEC. DecDEC achieves lower perplexity than Static while using 4× or even 8× fewer channels (e.g., $k_{\text{chunk}} = 32$ or 16 vs. 128), highlighting the effectiveness of incorporating input-awareness. Meanwhile, the perplexity gap between DecDEC and Exact is minimal, with almost overlap-

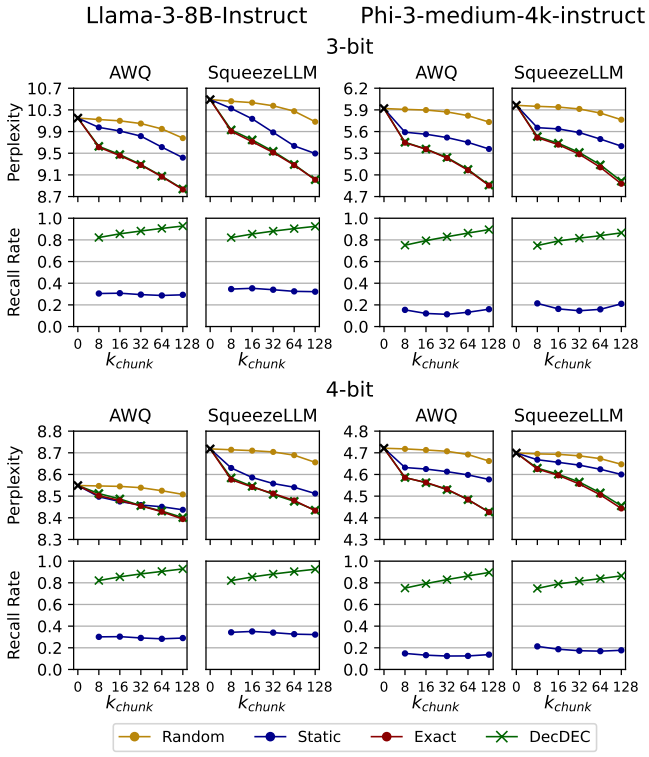


Figure 16: Comparison against random, static, and exact channel selection. Perplexity is shown on the top rows (lower is better), and recall is shown on the bottom rows (higher is better). The x markers in black correspond to baselines without DecDEC ($k_{chunk} = 0$).

ping curves. These results are explained by the recall rates: while DecDEC achieves around 80% recall relative to Exact, the recall of Static falls significantly short, at around 30% or below.

Impact of Residual Bitwidth. Table 2 presents the impact of residual bitwidth selection. In addition to the default 4-bit setting, we evaluate 2-bit, 8-bit, and full-precision (FP16) residuals with varying k_{chunk} for 3-bit models by perplexity on WikiText. Cells with the same color indicate cases where the total data transfer via PCIe is approximately equivalent. For example, $k_{chunk} = 8$ with 4-bit residuals requires a similar data transfer amount as the following combinations: $k_{chunk} = 16$ at 2-bit, $k_{chunk} = 4$ at 8-bit, and $k_{chunk} = 2$ at FP16. The best results within each color group are highlighted. Across all cases, a residual bitwidth of 4 either achieves the best or comes very close to it, supporting our default setting.

5.3 End-to-End Evaluation

Methodology. In this section, we present case studies demonstrating how DecDEC advances model quality while minimizing latency increases across three desktop GPUs—RTX 4090, 4080 Super (4080S), and 4070 Super (4070S)—as well as two

	Llama-3-8B-Instruct								Phi-3-medium-4k-instruct							
	AWQ 3-bit				SqueezeLLM 3-bit				AWQ 3-bit				SqueezeLLM 3-bit			
k_{chunk}	2-bit	4-bit	8-bit	FP16	2-bit	4-bit	8-bit	FP16	2-bit	4-bit	8-bit	FP16	2-bit	4-bit	8-bit	FP16
2				9.83				10.10				5.56				5.63
4			9.72	9.72			9.97	9.97			5.51	5.51			5.55	5.56
8		**9.63**	9.58	9.58		**9.93**	9.79	9.78		**5.45**	5.45	5.45		5.53	5.47	5.47
16	**9.63**	**9.47**	9.43	9.43	9.95	**9.75**	9.59	9.58	5.47	**5.36**	5.35	5.35	**5.51**	5.44	5.36	5.37
32	9.49	**9.29**	9.25	9.25	9.80	**9.54**	9.36	9.36	**5.36**	5.24	5.23	5.23	**5.41**	5.31	5.23	5.24
64	9.31	**9.07**	9.03		9.59	**9.29**	9.11		**5.22**	5.08	5.06		**5.28**	5.14	5.06	
128	9.12	**8.84**			9.36	**9.01**			**5.04**	4.86			**5.10**	**4.91**		
256	8.94				9.14				**4.83**				**4.91**			

Table 2: Impact of residual bitwidth. Lower is better.

laptop GPUs—RTX 4070 Mobile (4070M), and 4050 Mobile (4050M). The specifications for these GPUs are listed in Table 1. We run the tuner for DecDEC with four target slowdown rates (2.5%, 5%, 10%, 20%), and evaluate perplexity on WikiText as well as end-to-end inference latency with the resulting configurations. For the base GEMV kernel, we use LUTGEMM [23] for AWQ and Any-Precision LLM [45] for SqueezeLLM—both state-of-the-art kernels for uniform and non-uniform quantization, respectively. We integrate DecDEC into a PyTorch-based inference pipeline optimized with the torch.compile feature [5] and measure the average time taken per token generation over 1024 tokens.

For the 3.5-bit models, we do not run the tuner separately. Instead, we construct the configuration by combining the tuning results with the 3-bit and 4-bit models. Specifically, we use the configuration from tuning with the 3-bit model for layers quantized to 3 bits, and the configuration from tuning with the 4-bit model for layers quantized to 4 bits—both assuming the same target slowdown rate.

Results. Table 3 lists the configurations obtained from the tuner along with the actual end-to-end slowdowns compared to the baseline. We report only the results for 3-bit as similar trends are observed for the 3.5-bit and 4-bit configurations. In all cases, the actual slowdown is below the target rate. This is expected as the tuner configures parameters conservatively. The tuner targets only the kernel times of linear operations, while other operations outside the linear layers (e.g., attention, normalization) also contribute to overall inference time. Consistent with the observations in Section 5.1, in general, selected k values are higher for GPUs with greater PCIe-to-GPU memory bandwidth ratio (4050M $>$ 4070M $\simeq$ 4070S $>$ 4080S $>$ 4090).

Figure 17 shows the trends of end-to-end inference latency versus perplexity. All Phi-3 cases, as well as AWQ 3.5-bit, 4-bit, and SqueezeLLM 4-bit cases of Llama-3 face out-of-memory issues on the 4050M, and are thus excluded. Similarly, the AWQ 4-bit case of Phi-3 for 4070M is also excluded. In each line in Figure 17, the x marker represents the baseline, while subsequent markers show the results for DecDEC on the four target slowdown rates, in increasing order.

For all cases, DecDEC demonstrates promising Pareto-optimal trade-offs between model quality and inference latency. The results for target slowdown rates of 2.5% and 5% are particularly impressive, while 10%–20% show diminish-

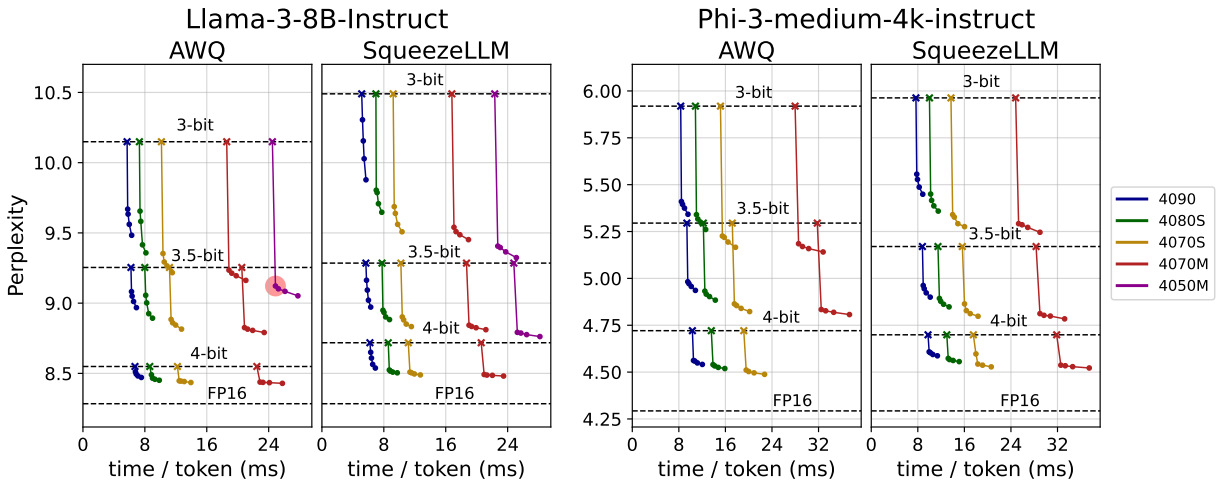


Figure 17: Perplexity against time per token on various NVIDIA GPUs. The x markers indicate baseline values, where DecDEC is not applied ($k_{chunk} = 0$). Subsequent markers show DecDEC results on target slowdown rates 2.5%, 5%, 10%, and 20%.

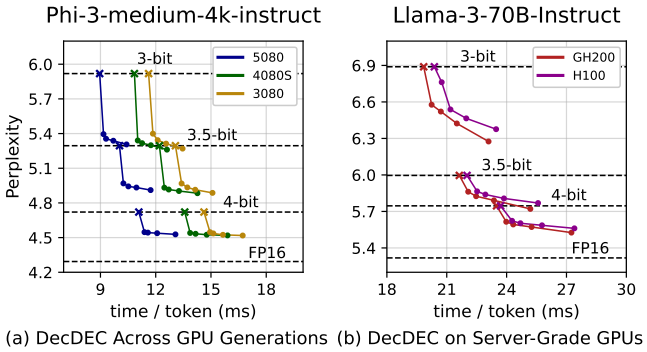
		Llama-3-8B-Instruct				Phi-3-medium-4k-instruct			
		AWQ 3-bit		SqueezeLLM 3-bit		AWQ 3-bit		SqueezeLLM 3-bit	
GPU	**Target**	**Tuner Results**	**Slowdown**	**Tuner Results**	**Slowdown**	**Tuner Results**	**Slowdown**	**Tuner Results**	**Slowdown**
4090	**2.5%**	24 / (4, 4, 8, 9)	1.3%	56 / (2, 0, 2, 2)	2.0%	20 / (8, 6, 13, 13)	1.5%	47 / (5, 2, 7, 10)	1.8%
	5%	24 / (5, 7, 9, 10)	2.2%	56 / (1, 1, 2, 6)	4.0%	47 / (10, 11, 13, 13)	3.8%	35 / (6, 6, 9, 10)	3.5%
	10%	24 / (10, 11, 11, 11)	4.9%	38 / (5, 4, 5, 7)	6.1%	35 / (15, 15, 15, 14)	7.8%	47 / (10, 11, 13, 11)	7.3%
	20%	24 / (15, 15, 16, 15)	10.4%	28 / (10, 10, 10, 10)	10.8%	20 / (18, 19, 18, 18)	15.0%	35 / (16, 16, 15, 14)	15.3%
4080S	**2.5%**	6 / (9, 5, 9, 6)	1.1%	24 / (10, 11, 14, 16)	0.6%	30 / (18, 17, 18, 20)	1.7%	30 / (14, 14, 15, 16)	2.0%
	5%	16 / (10, 10, 11, 9)	2.5%	24 / (14, 14, 14, 14)	2.2%	30 / (21, 21, 23, 21)	3.9%	30 / (17, 17, 20, 19)	3.7%
	10%	24 / (18, 20, 23, 18)	5.5%	24 / (18, 19, 18, 17)	4.7%	35 / (24, 25, 24, 24)	8.3%	35 / (22, 22, 23, 21)	7.6%
	20%	24 / (26, 27, 25, 24)	11.5%	38 / (23, 24, 22, 23)	10.8%	35 / (30, 32, 28, 29)	16.2%	35 / (27, 27, 25, 25)	15.1%
4070S	**2.5%**	28 / (25, 26, 26, 26)	2.0%	19 / (16, 15, 24, 21)	1.5%	20 / (31, 31, 37, 32)	2.1%	20 / (25, 26, 30, 28)	2.1%
	5%	24 / (31, 31, 35, 29)	3.7%	24 / (21, 22, 24, 24)	3.0%	20 / (35, 35, 37, 34)	4.2%	20 / (30, 30, 30, 29)	4.2%
	10%	24 / (35, 35, 36, 34)	7.1%	24 / (28, 30, 30, 27)	6.2%	24 / (39, 40, 41, 38)	8.5%	24 / (34, 35, 35, 33)	8.6%
	20%	28 / (44, 44, 41, 42)	13.9%	24 / (35, 36, 34, 35)	12.4%	20 / (45, 46, 44, 45)	17.0%	20 / (40, 41, 38, 39)	16.9%
4070M	**2.5%**	16 / (38, 39, 39, 40)	1.7%	12 / (28, 28, 35, 29)	1.6%	18 / (41, 41, 41, 42)	2.1%	17 / (33, 35, 35, 34)	2.1%
	5%	16 / (42, 43, 42, 42)	3.4%	16 / (33, 34, 36, 33)	3.1%	18 / (44, 46, 43, 44)	4.4%	18 / (37, 38, 37, 37)	4.3%
	10%	16 / (46, 46, 44, 45)	6.6%	16 / (37, 38, 37, 37)	6.1%	18 / (46, 48, 46, 46)	8.3%	18 / (40, 41, 39, 39)	8.5%
	20%	16 / (50, 50, 49, 50)	13.4%	16 / (42, 42, 41, 41)	12.7%	18 / (53, 53, 50, 51)	17.2%	18 / (44, 44, 43, 43)	16.7%
4050M	**2.5%**	8 / (55, 56, 58, 55)	1.7%	7 / (45, 45, 50, 44)	1.7%	OOM	-	OOM	-
	5%	8 / (59, 59, 59, 58)	3.3%	9 / (48, 48, 48, 48)	3.3%	OOM	-	OOM	-
	10%	10 / (62, 62, 62, 62)	6.6%	10 / (52, 53, 52, 52)	6.3%	OOM	-	OOM	-
	20%	10 / (70, 70, 68, 68)	13.5%	10 / (59, 60, 57, 58)	12.6%	OOM	-	OOM	-

Table 3: Tuner results ($n_{tb}^{max}/(k_{chunk}^{qkv}, k_{chunk}^{o}, k_{chunk}^{gu}, k_{chunk}^{d})$) for four target slowdown rates (2.5%, 5%, 10%, 20%) and corresponding actual slowdown rates for 3-bit Llama-3 and Phi-3. Phi-3 is out of memory (OOM) on the 4050M GPU.

ing returns. On platforms with high PCIe-to-GPU memory bandwidth ratios, such as the 4070S, 4070M, and 4050M, DecDEC at 2.5% slowdown on 3-bit models sometimes outperforms 3.5-bit baselines. In these cases, DecDEC yields *Pareto-dominant* solutions by excelling in model quality, latency, and memory. Examples include AWQ Llama-3 (4070M, 4050M) and AWQ Phi-3 (4070S, 4070M). A particularly noteworthy case is the AWQ 3-bit Llama-3 on 4050M, where DecDEC reduces perplexity from 10.15 to 9.12 with only a 1.7% latency slowdown (highlighted by a red circle). This outperforms the 3.5-bit baseline, which is infeasible on 4050M without DecDEC due to memory limits. This case highlights DecDEC's effectiveness in pushing the boundaries of quantized LLMs within memory capacity constraints.

5.4 Evaluation across GPU Generations

We evaluate the robustness of DecDEC across GPU generations, as both GPU memory bandwidth and PCIe bandwidth, two key factors influencing its effectiveness, may increase with newer architectures. To this end, we use three consumer-grade GPUs from the same product class across different



(a) DecDEC Across GPU Generations (b) DecDEC on Server-Grade GPUs

Figure 18: Perplexity vs. time per token: (a) across different GPU generations and (b) on server-grade GPUs.

	Memory Bandwidth	PCIe Bandwidth	R_{bw}
RTX 5080	960 GB/s	64 GB/s	15
RTX 4080S	736 GB/s	32 GB/s	23
RTX 3080	760 GB/s	32 GB/s	24

Table 4: 80-class GPU specifications across generations.

generations—RTX 3080, 4080S, and 5080—whose specifications are listed in Table 4. The R_{bw} values (lower is better) remain essentially unchanged from the 3080 to the 4080S and even decrease from the 4080S to the 5080. Figure 18(a) plots the perplexity of AWQ-quantized Phi-3 with DecDEC augmentation against end-to-end latency on these GPUs, using the methodology described in Section 5.3. The improvements delivered by DecDEC are comparable across all three cards, confirming its robustness across generations.

5.5 Applicability to Server-Grade GPUs

Although DecDEC primarily targets single-batch inference on client (edge) devices, this section evaluates its effectiveness on server-grade GPUs. We measure perplexity versus end-to-end latency using an AWQ-quantized Llama-3-70B-Instruct model augmented with DecDEC on the H100 SMX5 and GH200, following the methodology described in Section 5.3. Both GPUs provide 3.36 TB/s of memory bandwidth; however, the GH200's 450 GB/s NVLink-C2C far exceeds the H100's 64 GB/s PCIe, yielding a much lower R_{bw}.

Figure 18(b) shows the results. DecDEC improves perplexity on both GPUs with minimal latency overhead. However, the GH200's advantage is smaller than the R_{bw} gap suggests. DecDEC assumes a DRAM-bound GEMV, where reallocating SMs for error compensation do not impact GEMV latency. On server GPUs, however, quantized GEMV (e.g., LUTGEMM [23]) is L1-bound, not DRAM-bound. Since L1 throughput scales with active SMs, reallocating SMs increases GEMV latency, limiting DecDEC's benefit despite low R_{bw}. Enhancing quantized GEMV kernels for server-grade GPUs by mitigating L1 bottlenecks could unlock further gains.

6 Related Work

GPU Implementations for Weight-Only Quantization. Numerous studies have proposed efficient GPU implementations tailored for weight-only quantization of LLMs. LUTGEMM [23] replaces the expensive dequantization with simple LUT operations. Marlin [20] supports FP16-INT4 GEMM across various batch sizes, and Quant-LLM [63] introduces a GPU kernel that efficiently handles non-power-of-2 bitwidths (e.g., 6-bit). FLUTE [24] provides a specialized kernel for non-uniform quantization. Any-Precision LLM [45] suggests a memory-efficient kernel for adaptive bitwidth selection. These implementations can seamlessly integrate with DecDEC, benefiting from its dynamic error compensation.

LLM Inference with External Memory. Several works address GPU memory limitations in LLM inference by leveraging external memories like CPU memory or disk. DeepSpeed-Inference [4] and FlexGen [50] focus on throughput-oriented, out-of-core LLM inference. Pre-gated MoE [29] retrieves the parameters of only the activated experts in MoE models from CPU memory. LLM-in-a-Flash [3] introduces a flash memory-based inference system exploiting activation sparsity. InfiniGen [35] offloads the key-value cache to CPU memory. PowerInfer [51] proposes a GPU-CPU hybrid inference engine that leverages activation sparsity in ReLU-based models. Though these approaches share the same goal as DecDEC in aiming to extend GPU memory, they address distinct scenarios with different challenges and opportunities.

LLM Compression Methods. In addition to quantization, pruning can improve the efficiency of LLM inference [18, 38, 53]. Other techniques, including knowledge distillation [22, 26] and low-rank decomposition [48, 67], offer additional ways to compress LLMs. These methods are orthogonal to quantization and, consequently, to DecDEC.

7 Conclusion

We propose DecDEC, an inference scheme for low-bit LLMs that improves model quality by correcting quantization errors through selective retrieval of residuals stored in CPU memory. By focusing on dynamically identified salient channels at each decoding step, DecDEC maximizes error compensation within a limited transfer volume. DecDEC significantly improves quality of quantized LLMs with minimal overheads.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (RS-2024-00340008, RS-2024-00405857). Jae W. Lee is the corresponding author.

References

[1] NVIDIA H100 Tensor Core GPU. https://resources.nvidia.com/en-us-data-center-overview-mc/en-us-data-center-overview/nvidia-tensor-core-gpu-datasheet, 2024.

[2] Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *ACM J. Exp. Algorithmics*, 2012.

[3] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S. Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. LLM in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2022.

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

[6] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Quantizable transformers: Removing outliers by helping attention heads do nothing. In *Advances in Neural Information Processing Systems*, 2023.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

[8] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[9] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. QuIP: 2-bit quantization of large language models with guarantees. In *Advances in Neural Information Processing Systems*, 2023.

[10] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2024.

[11] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[12] Tim Dettmers, Ruslan A. Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. SpQR: A sparse-quantized representation for near-lossless LLM weight compression. In *The Twelfth International Conference on Learning Representations*, 2024.

[13] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[14] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. In *Advances in Neural Information Processing Systems*, 2020.

[15] Zhen Dong, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *2019 IEEE/CVF International Conference on Computer Vision*, 2019.

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

[17] Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

[18] Elias Frantar and Dan Alistarh. Sparsegpt: massive language models can be accurately pruned in one-shot. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[19] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023.

[20] Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025.

[21] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

[22] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[23] Minsub Kim Sungjae Lee Jeonghoon Kim Beomseok Kwon Se Jung Kwon Byeongwook Kim Youngjoo Lee Gunho Park, Baeseong Park and Dongsoo Lee. Lutgemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[24] Han Guo, William Brandon, Radostin Cholakov, Jonathan Ragan-Kelley, Eric P. Xing, and Yoon Kim. Fast matrix multiplications for lookup table-quantized LLMs. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024.

[25] Jung Hwan Heo, Jeonghoon Kim, Beomseok Kwon, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. Rethinking channel dimensions to isolate outliers for low-bit weight quantization of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[26] Cheng-Yu Hsieh, Chun-Liang Li, Chih-kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, 2023.

[27] Wei Huang, Haotong Qin, Yangdong Liu, Yawei Li, Xianglong Liu, Luca Benini, Michele Magno, and Xiaojuan Qi. Slim-llm: Salience-driven mixed-precision quantization for large language models. 2024.

[28] Wei Huang, Xingyu Zheng, Xudong Ma, Haotong Qin, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu, and Michele Magno. An empirical study of llama3 quantization: From llms to mllms, 2024.

[29] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. Pregated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture*, 2024.

[30] Jeonghoon Kim, Jung Hyun Lee, Sungdong Kim, Joonsuk Park, Kang Min Yoo, Se Jung Kwon, and Dongsoo Lee. Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization. In *Advances in Neural Information Processing Systems*, 2023.

[31] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. SqueezeLLM: Dense-and-sparse quantization. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[33] Changhun Lee, Jungyu Jin, Taesu Kim, Hyungjun Kim, and Eunhyeok Park. Owq: Outlier-aware weight quantization for efficient fine-tuning and inference of large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

[34] Janghwan Lee, Minsoo Kim, Seungcheol Baek, Seok Hwang, Wonyong Sung, and Jungwook Choi. Enhancing computation efficiency in large language models

through weight and activation quantization. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[35] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[36] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In *Proceedings of Machine Learning and Systems*, 2024.

[37] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. LLM-QAT: Data-free quantization aware training for large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024.

[38] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. In *Advances in Neural Information Processing Systems*, 2023.

[39] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[40] Microsoft. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[41] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. In *Proc. VLDB Endow.*, 2021.

[42] NVIDIA. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2024.

[43] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024.

[44] OpenAI. Gpt-4 technical report, 2024.

[45] Yeonhong Park, Jake Hyun, SangLyul Cho, Bonggeun Sim, and Jae W. Lee. Any-precision llm: Low-cost deployment of multiple, different-sized llms. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

[46] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. Evaluating characteristics of cuda communication primitives on high-bandwidth interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019.

[47] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 2020.

[48] Varun Srivastava Rajarshi Saha and Mert Pilanci. Matrix Compression via Randomized Low Rank and Low Precision Factorization. In *Advances in Neural Information Processing Systems*, 2023.

[49] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. 2023.

[50] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[51] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024.

[52] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. 2022.

[53] Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models. 2023.

[54] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, , and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. 2022.

[55] Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

[56] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models, 2023.

[57] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP$\#$: Even better LLM quantization with hadamard incoherence and lattice codebooks. In *Forty-first International Conference on Machine Learning*, 2024.

[58] turboderp. ExLlamaV2. https://github.com/turboderp/exllamav2.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[60] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. Efficient large language models: A survey, 2024.

[61] Zhe Wang, Jie Lin, Xue Geng, Mohamed M. Sabry Aly, and Vijay Chandrasekhar. Rdo-q: Extremely fine-grained channel-wise quantization via rate-distortion optimization. In *Computer Vision – ECCV 2022*, 2022.

[62] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2024.

[63] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. Quant-LLM: Accelerating the serving of large language models via FP6-Centric Algorithm-System Co-Design on modern GPUs. In *2024 USENIX Annual Technical Conference*, 2024.

[64] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[65] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In *Advances in Neural Information Processing Systems*, 2022.

[66] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, 2022.

[67] Zhihang Yuan, Yuzhang Shang, Yue Song, Qiang Wu, Yan Yan, and Guangyu Sun. Asvd: Activation-aware singular value decomposition for compressing large language models, 2023.

[68] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2024.

[69] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models, 2024.



Stripeless Data Placement for Erasure-Coded In-Memory Storage

Jian Gao[†] Jiwu Shu[*†] Bin Yan[†] Yuhao Zhang[†] Keji Huang[‡]

[†]*Tsinghua University*
[‡]*Huawei Technologies Co., Ltd*

Abstract

Erasure coding plays a crucial role in distributed storage systems to provide fault tolerance at a low storage cost. Conventional erasure coding schemes determine data placement based on stripes. However, placing data into stripes can incur non-negligible performance overheads that will manifest in emerging fast in-memory storage systems, making conventional erasure coding schemes suboptimal in such scenarios.

Aiming to eliminate such overheads, we present Nos, a *stripeless* placement scheme for erasure-coded in-memory storage. It lets each node *independently* replicate data to other nodes and encode received data replicas into parities with XOR. Thus, it avoids the overheads caused by stripes. To enable failure recovery, Nos uses a combinatoric structure called *symmetric balanced incomplete block design* (SBIBD) to decide primary-to-backup node affinities during replication. Atop Nos, we further build Nostor, a distributed in-memory key-value store. Evaluations demonstrate that Nostor achieves 1.61× to 2.60× throughputs with similar or lower latencies than stripe-based erasure coding baselines.

1 Introduction

Fault tolerance is crucial for emerging distributed in-memory storage systems whose nodes can experience multiple types of failures [24]. These systems, built atop the fast Remote Direct Memory Access (RDMA) networks, provide orders of magnitude faster access to numerous latency-sensitive hot data objects than disks [6,7,13,26,30,53]. However, they still need to reload data from the underlying slow storage upon failure recovery, during which the system can experience significant performance degradations [15,25,40] or data unavailability. Hence, in-memory storage systems need fault tolerance to preserve availability and performance during partial failures.

Redundancy is the key to fault tolerance. There are two major approaches to achieving data redundancy: replication and erasure coding. Replication is well-known for its simplicity but also high storage overhead. Erasure coding offers a tradeoff: it has much less storage overhead than replication, which comes at a cost in performance by incurring extra computation. Since main memory is significantly more expensive than disks, erasure coding becomes an appealing fault tolerance solution for in-memory storage systems as it allows accommodating much more objects than replication [47,56].

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

Stripe is a basal concept of virtually all existing erasure coding schemes. It consists of a set of chunks, some of which are data chunks and the others are parity chunks. Stripe requires that every piece of data must be assigned to *exactly one stripe*, and all methods to perform such assignments fall in two categories: intra-object [9,25,27] and inter-object [6,55]. Intra-object means that a stripe contains only one object, while inter-object means that a stripe contains multiple objects.

However, we find that on the fast main memory, the stripe concept tarnishes the performance-storage tradeoff by incurring high costs in the storage systems when they need to determine data placement. Here, existing erasure coding schemes either pay more on performance or save less memory:

- With intra-object methods, they must split objects into chunks, bringing significant overheads due to the high I/O fanout when accessing the objects.
- With inter-object methods, they either deploy a metadata service (MDS) to group objects into stripes, or they assign objects to stripes following a statically determined policy. The former introduces the MDS as a centralized bottleneck, while the latter wastes memory as there is no guarantee that objects are evenly distributed across stripes without any centralized coordination.

The main memory is fast and expensive, so we can afford the costs of neither choices. Still, the stripe concept forces us to choose one, putting us into a dilemma.

To circumvent these drawbacks, abandoning stripes seems like a good idea. However, precedent research has never taken this approach. Stripes are critical when they need to tolerate multiple chunk failures: failed chunks (within a threshold) can always be expressed as linear combinations of other living chunks and, thus, get recovered. Without stripes, this property might not hold, causing data loss risks.

In this paper, however, we argue that with a proper data placement policy, it is possible to tolerate multiple chunk failures without stripes. Thus, we can avoid stripe's drawbacks and make erasure coding's performance-storage tradeoff favorable for in-memory storage systems again. Without stripes, each node can independently replicate data chunks on it to other nodes; at the same time, they independently encode received data chunk replicas into their XOR sums, i.e., parity chunks. This greatly simplifies the write path.

Guided by the principle above, we present Nos, a *stripeless* data placement scheme for erasure coding. Like the famous

Reed-Solomon code [37], Nos has two parameters (k, p):

- Parameter p determines the fault tolerance threshold. Each data chunk is replicated for $(p+2)$ times – one primary and $(p+1)$ backups, to tolerate p node failures.
- Parameter k determines the storage overhead. Each node independently XORs k data chunk replicas into a parity chunk, bringing a storage amplification factor of $(p+1)/k$.

The key of Nos is that it selects backup nodes for each primary node according to a combinatoric structure called *symmetric balanced incomplete block design* (SBIBD). We prove that this model guarantees a successful recovery of any data chunk upon any p node failures. The amortized cost to reconstruct a lost chunk is no more than reading k chunks, also similar to the Reed-Solomon code.

Atop Nos, we build Nostor, a distributed key-value store. Nostor's server nodes store objects in their main memory and communicate via remote procedure calls (RPC) atop RDMA. Nostor employs Nos flexibly to achieve both low latency and high throughput. It also ensures data consistency for common-case I/Os, degraded I/Os, and node repairs with a versioning-based approach.

We implemented Nostor in Rust and make it open-source at GitHub. Our evaluation shows that, under real-world workloads, Nostor outperforms conventional stripe-based erasure coding schemes with 1.61× to 2.60× throughput improvements and similar or lower latencies. It consumes 18.7% to 57.4% less memory than primary-backup replication but is often as performant as it. Nostor also improves the node repair time by 16.4%. Nevertheless, as a performance trade-off, Nostor's degraded read latency can be 35.0% to 62.4% higher than conventional erasure coding in the worst case.

2 Background

2.1 Erasure Coding and Stripes

Erasure coding is an established data redundancy mechanism. Many distributed storage systems, both experimental [17, 41, 42] and productional [16, 18, 23, 32, 43, 50], have erasure coding enabled or provide it as an option for ensuring fault tolerance for data in them. Stripe is the fundamental concept of virtually all existing erasure coding schemes. We use the Reed-Solomon code [37], one of the most widely adopted coding schemes, as an example to elaborate on.

The Reed-Solomon code has two parameters: (k, p). Both are positive integers. The parameter p determines the fault tolerance threshold: the coding scheme can tolerate up to p simultaneous failures. The parameter k governs the storage overhead: the storage amplification factor is p/k.

Specifically, the Reed-Solomon code partitions all data chunks into groups of size k. Within each group, it multiplies a $p \times k$ encode matrix to the k data chunks to get p parity chunks. These k data and p parity chunks together form a stripe. The stripe can recover any p failed chunks by multiplying an inversion of the encode matrix to the remaining chunks.

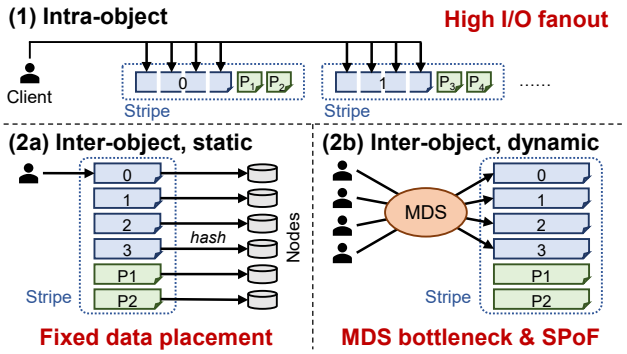


Figure 1: Summary of the drawbacks of stripes.

An essential characteristic of the Reed-Solomon code is that each chunk belongs to *exactly one* stripe. This characteristic is universal: it applies to all existing erasure coding schemes despite their diverse designs. Specifically, each data chunk must be assigned to a stripe, and each parity chunk originates only from the data chunks within the same stripe.

2.2 Drawbacks of Stripes

Stripe is the basic concept of erasure coding in conventional disk-based storage systems. However, when we move to the fast in-memory storage, its drawbacks start to manifest. The root cause is that *data placement is limited by stripes*. With a stripe-based coding scheme, determining data placement it is a difficult task that no existing solution can handle perfectly.

We still take the Reed-Solomon code as an example and focus on how it determines data placement in practice in an erasure-coded key-value store. There are two types of placement policies: intra-object and inter-object. We discuss them below and summarize their drawbacks in Figure 1.

2.2.1 Intra-object

Each object is first split into k chunks (each placed on a different node) and takes up an entire stripe. Data from different objects will not mix together in the same stripe. Examples include Ceph [50], EC-Cache [36], Giza [8], and Hydra [25].

For data availability, chunks in a stripe must be placed on different nodes, which means that reading or writing an object requires contacting k or $(k+p)$ nodes. Such a high I/O fanout can put significant strain on the network and cut down the maximum throughput proportionally (§6.2). Also, recent analyses revealed that small objects are prevalent in real-world workloads on in-memory key value stores [54]. It is unacceptable to access multiple nodes only to read/write a value that is smaller than 1 KB in around 80% of the cases [5].

2.2.2 Inter-object

Objects encode with each other; each stripe contains k objects. Examples include Cocytus [6] and LogECMem [9]. Here, we must assign k objects on different nodes to a stripe. There are two common assignment policies: static and dynamic.

Static assignment. For any object, we decide its belonging stripe only by its metadata (e.g., key hash) with a static policy. A significant drawback of this policy is that it does not respect the storage system's current state. In practice, this means that we cannot adapt to real-time placement constraints. For example, some nodes may suffer from temporary slowdowns, which can affect a wide range of user requests accessing objects whose primaries or parities are stored on these nodes.

Hash-based policies are also prone to the storage space waste caused by the empty chunks in stripes. It might, for example, hash objects 0, 1, 2, and 3 into the same stripe, but objects 1, 2, and 3 may never exist in the workload. This can cause the memory overheads to far exceed our expectations. Cocytus [6] alleviates this problem by manually allocating objects and encoding at virtual address level. However, in exchange, its allocation policy brings performance overheads (§6.2) and can cause memory wastes (§6.5).

Dynamic assignment. For any object, we dynamically decide its belonging stripe and placement at runtime, usually by consulting a metadata service (MDS) or proxy [9,27,55]. The MDS can choose a proper node (e.g., with the lowest load) to store the object and group it with other existing objects into a stripe, thus circumventing the drawbacks of static policies.

The problem is that we do not know an object's placement in advance, so every attempt to access the object has to first contact the MDS to get its metadata. This not only adds a network roundtrip's latency to the read/write critical paths but also makes the MDS a bottleneck and a single point of failure (SPoF). While replicating the MDS [9] solves the SPoF, it again aggravates the performance bottleneck due to the overheads brought by replication. Also, because objects are often small, the memory overheads brought by the (possibly replicated) MDS are non-negligible since the metadata size can be comparable to the object itself.

2.2.3 Summary

Stripes forces us to assign objects to them. However, every possible assignment scheme has significant drawbacks, making the stripe concept itself problematic.

3 Motivation

3.1 Abandoning Stripes

Stripe is the root cause of all the problems described in the previous section. Therefore, we can immediately solve them by abandoning the stripe concept. This is an unprecedented idea by now, but we can imagine how an erasure-coded storage system without stripes will look like:

- For availability, each object is stored on one *primary node* and replicated to multiple *backup nodes*. The primary node can independently and dynamically determine the object's backup nodes without having to consult a separate MDS.
- For low memory overhead, each node encodes received object replicas into parities. It can simply XOR the objects together instead of multiplying them with some complicated encode matrix. To recover a specific object, we find the parity encoding it, read all other objects encoded in the parity, and XOR them up to reconstruct the object.

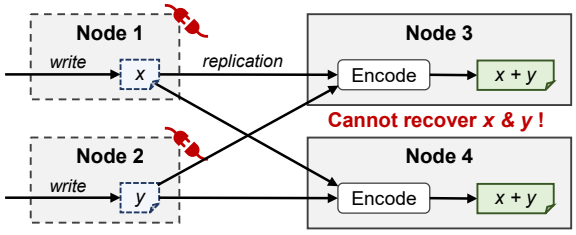


Figure 2: Demonstration of a strawman stripeless placement scheme. It risks data loss because it replicates data from two primary nodes to the same two backup nodes.

The imagination above is intuitively correct when only one node fails. But how can we perform failure recovery when multiple nodes fail simultaneously? Take the case in Figure 2 as an example. Objects x and y are stored on primary nodes 1 and 2, respectively, and they are both replicated to nodes 3 and 4 in case of two node failures. However, the system still risks data loss because if nodes 3 and 4 both XOR these objects together into $P_1 = P_2 = x \oplus y$, when nodes 1 and 2 fail, we cannot decode x and y from the remaining $x \oplus y$.

We find that the cause of this problem is that x and y, with different primary nodes, are replicated to the same two backup nodes 3 and 4. To eliminate this problem, we can assign different backup nodes for different primary nodes, so that *if two objects have different primary nodes, they cannot be replicated to the same two backup nodes.*

Formally, for a (symmetric) cluster of v nodes, we define a *primary-backup affinity matrix* $A_{v\times v}$ whose every element is 0 or 1. $A_{ij} = 1$ iff node i can replicate its primary objects to node j. Our requirement is: for any two rows r_1, r_2, there may exist at most one column c such that $A_{r_1 c} = A_{r_2 c} = 1$.

The final problem that remains is how to find the matrix A. Fortunately, it turns out that combinatorial mathematicians have already found it for us. Specifically, A should be a *symmetric balanced incomplete block design* (SBIBD). Below, we briefly introduce SBIBD and explain its connection with our stripeless data placement scheme.

3.2 SBIBD Primer

SBIBD is a combinatoric structure that was born in the 19th century [21,22,45] and can be found in most combinatorics textbooks today [4]. We only introduce necessary parts of the SBIBD theory here; more can be found in the textbooks.

SBIBD has three parameters: (v,k,λ). A (v,k,λ)-SBIBD is a 0/1 matrix $A_{v\times v}$. It has exactly k ones in each row and column; also, for any two rows r_1, r_2 in A, we have:

$$\left|\left\{c \mid A_{r_1 c} = A_{r_2 c} = 1\right\}\right| = \lambda$$

i.e., there are exactly λ column(s) with ones at both rows. We are particularly interested in the case where $\lambda = 1$, in which our requirement described in §3.1 is innately satisfied.

$(v,k,1)$-SBIBD has an important implication: v and k are not independent parameters. To show this, consider the number of ways to pick two rows r_1, r_2 and a column c such that $A_{r_1c} = A_{r_2c} = 1$. Since every column has k ones, this number is obviously $v\binom{k}{2}$. Also, by definition, for every distinct row pair (r_1, r_2) there exist exactly one such c, so the number is also $\binom{v}{2}$. Therefore,

$$\binom{v}{2} = v\binom{k}{2} \quad \Rightarrow \quad v = k^2 - k + 1 \tag{1}$$

i.e., the SBIBD must be a $(k^2-k+1, k, 1)$-SBIBD. We redefine $v = k^2 - k + 1$ hereinafter for convenience.

$(v,k,1)$-SBIBDs exist and can be efficiently computed for all $k = q+1$, where q is a prime power. As a result, they exist for most commonly-used k values in erasure coding, e.g., 3, 4, 5, 6, 8, 10, 12, and 14. To construct such a SBIBD, we first construct its topmost row r_1 and then cyclic-rotate it by $(i-1)$ units to get the row r_i [44]. Since SBIBD does not require an order among the rows in A, we can choose r_1 properly such that $A_{11} = 0$. This will ensure that A has a zero main diagonal. For example, when $k = 4, v = 13$, an SBIBD constructed by the method above is shown as follows.

	1	2	3	4	5	6	7	8	9	10	11	12	13
r_1	0	0	0	1	0	0	0	0	0	1	0	1	1
r_2	1	0	0	0	1	0	0	0	0	0	1	0	1
r_3	1	1	0	0	0	1	0	0	0	0	0	1	0
r_4	0	1	1	0	0	0	1	0	0	0	0	0	1
r_5	1	0	1	1	0	0	0	1	0	0	0	0	0
r_6	0	1	0	1	1	0	0	0	1	0	0	0	0
r_7	0	0	1	0	1	1	0	0	0	1	0	0	0
r_8	0	0	0	1	0	1	1	0	0	0	1	0	0
r_9	0	0	0	0	1	0	1	1	0	0	0	1	0
r_{10}	0	0	0	0	0	1	0	1	1	0	0	0	1
r_{11}	1	0	0	0	0	0	1	0	1	1	0	0	0
r_{12}	0	1	0	0	0	0	0	1	0	1	1	0	0
r_{13}	0	0	1	0	0	0	0	0	1	0	1	1	0

For example, for r_1, r_2, no column except $c = 13$ satisfies $A_{1c} = A_{2c} = 1$, verifiying SBIBD's property.

4 Nos

Nos[1] is an stripeless and inter-object data placement scheme for erasure coding, which is designed following the ideas discussed in the previous section. It tolerates $p < k$ failures in a cluster of size $v = k^2 - k + 1$. We first present its design (§4.1) and give it a demonstration (§4.2). Then, we briefly discuss its benefits and limitations (§4.3).

[1]The name stands for "no-stripe".

4.1 Code Design

Consider a cluster of v server nodes numbered $1, 2, \ldots, v$. Nos views a $(v,k,1)$-SBIBD as its primary-to-backup affinity matrix, which is in correspondence with our discussion in §3. Specifically, for an object whose primary is node i, we can send its replica to node j iff $A_{ij} = 1$. For any specific node j, the set $\{i \mid A_{ij} = 1\}$ is called its *replication source* set. Conversely, for any specific node i, the set $\{j \mid A_{ij} = 1\}$ is called its *replication target* set. Both sets have sizes of k.

Like the Reed-Solomon code, Nos has two parameters (k, p). Parameter k represents how many objects we encode into a parity and is co-dependent with the cluster size v following Equation (1). Parameter p represents the fault tolerance threshold. As a precondition, we require $k > p$. Most real-world storage systems naturally meet this precondition because $p \geq k$ would result in a relative storage overhead of at least 100%, which is usually undesirable.

The storage cluster must ensure data availability despite no more than p node failures. To this end, Nos stores each object on one primary node in its original form and encodes it with other objects on $(p+1)$ backup nodes using the following two-step erasure coding policy:

- The primary node replicates an object to nodes in its replication target set. There are k choices, and the primary node may select any $(p+1)$ out of them. This step is virtually equivalent to $(p+2)$-way replication. Since $A_{ii} = 0$ (zero main diagonal), object replicas are guaranteed not to be sent back to their primary nodes.
- Each node can receive object replicas from k other nodes. These replicas are temporarily buffered. In the background, when there is an object replica from every replication source node, the node collects one from each replication source and replaces them with their XOR sum, i.e., a parity.

Since the primary copy is the unchanged original data, Nos should be viewed as a systematic erasure coding scheme [29].

Primary nodes serve reads to the corresponding objects. If there are failed nodes, clients cannot directly access an object – let us call it x – whose primary node has failed. They need to contact living backup nodes of x to retrieve the parities and recover x. Each parity P of x is the XOR sum of x and other $(k-1)$ objects $x_1, x_2, \ldots, x_{k-1}$. If, fortunately, all these $(k-1)$ objects are alive, then the client can read them to decode x:

$$x = P \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{k-1}$$

We say x is *directly* recoverable by P in this case because P encodes only one failed object, that is, x.

However, when there are many failed nodes (no fewer than $(p+3)/2$, to be precise), every living parity P of x might encode some failed object y other than x itself, making direct recoveries impossible. In that case, we need to recursively recover y to finally recover x with P.

The key point is that because we select backup nodes according to a $(v,k,1)$-SBIBD, it is guaranteed that there exists

a P such that y is directly recoverable with it. As a result, even if we cannot recover x directly, we can still recover x in two steps: first, recover y; second, recover x.

Proof. Without loss of generality, let node 1 be the primary node of object x. Let $\mathcal{P}$ be the set of backup nodes where x is replicated to; $|\mathcal{P}| = p+1$.

To start recovery, node 1 must fail first. Each failed node in $\mathcal{P}$ can prevent up to two parities of x from being able to directly recover it (which we denote as *disabling* a parity below). For example, if $u \in \mathcal{P}$ fails, then a parity on u gets lost, and another parity that encodes x and an object from u now cannot directly recover x; by SBIBD's property, there cannot exist a third parity that encodes x and an object from u. This means that at least $1+(p+1)/2 = (p+3)/2$ failures are necessary to render x not directly recoverable.

Suppose we make f nodes in $\mathcal{P}$ fail such that x is no longer directly recoverable; let $\mathcal{F}$ be the set of these nodes. x now has $(p-f+1)$ living parities encoding it. Let $\mathcal{Y}$ be the set of failed objects other than x encoded in these parities. Consider any $y \in \mathcal{Y}$; let v be the primary node of y and P_{xy} be the parity that encodes both x and y. Object y also has $(p+1)$ parities encoding it; one parity might be on node 1 and get lost, and f parities are disabled by objects replicated from nodes in $\{1\} \cup \mathcal{F} - \{v\}$. By SBIBD's property, nodes in $\mathcal{F}$ except the one storing P_{xy} do not accept objects from node v. Therefore, there are $(p-f)$ remaining parities of y.

We still have a quota of $(p-f-1)$ further node failures. To prevent the recovery of x, we must select $(p-f+1)$ objects from $\mathcal{Y}$, one for each living parity of x, and make them not directly recoverable. These objects have $(p-f+1)(p-f)$ non-overlapping parities available. Similar to the discussion above, by SBIBD's property, each failed node not in $\{1\} \cup \mathcal{F}$ can disable up to two parities of a certain $y' \in \mathcal{Y}$; at the same time, it can also disable one parity for every other object in $\mathcal{Y}$. Hence, using the remaining node failure quota, we can disable up to $(p-f-1)(p-f+2)$ parities for the $(p-f+1)$ objects mentioned above. Because

$$(p-f-1)(p-f+2) < (p-f+1)(p-f)$$

we conclude that there exist at least one living parity of x, in which all failed objects except x are directly recoverable. Therefore, x is also recoverable. □

The proof's idea can also explain why Nos replicates objects to $(p+1)$ backup nodes. Intuitively, to tolerate p failures, each object should only replicate to p backup nodes. However, it can be shown that this can make the degraded read recursion infinite. An extra replica solves this problem.

4.2 Demonstration with $k = 4$

We demonstrate Nos with an example. Here, $k = 4$, $v = 13$, and we use the same SBIBD shown in §3.2. We assume that node 1 has failed, and a client now wants to access an object x whose primary node is node 1. Nodes 4, 10, 12, and 13 can accept backups of objects from node 1 and hold parities that encode x. Since we require $k > p$, p can only be 1, 2, or 3.

- When $p = 1$, the recovery is trivial: we can use any parity that encodes x since it cannot encode other failed objects.
- When $p = 2$, there can be one failed node other than node 1, which can prevent up to two parities from being able to directly recover x (e.g., if node 4 fails, the parity on it gets lost, and the parity on node 13 encoding an object from node 4 is also affected). Since we have three parities for x, one will be unaffected and can directly recover x.
- When $p = 3$, exhaustive search verifies that x is directly recoverable except when nodes 4 and 12 fail. In that case, x has two living parities: P_1 on node 10 and P_2 on node 13. They can both decode x, and we only discuss P_1 here for example. Besides x, P_1 encodes a failed object y from node 12, which we can directly recover with the parity on node 8 (encoding objects from nodes $\{5, 9, 10, 12\}$). Therefore, we can first decode y and then use P_1 to decode x.

4.3 Discussions

Nos aims at solving the drawbacks of stripes described in §2.2. It offers flexibility in data placement without an MDS by allowing the selection of backup nodes when replicating objects. It replaces matrix computation with XOR compared with the Reed-Solomon code, and hides parity encoding overheads in the background. The most important advantage is that despite the complex SBIBD theory behind it, Nos is easy to implement and deploy: primary nodes simply replicate data to backup nodes, and backup nodes encode received data into parities, both without any coordination with other nodes.

Nevertheless, such advantages come with some limitations, which we list and discuss below.

Ineffectiveness with slow storage. The goal of Nos is to accelerate fast in-memory storage systems. For conventional storage systems built atop slow disks, Nos can have limited benefits since the disks dominate their performance.

Incompatibility with large k values. Some code schemes, like the wide-stripe erasure codes [17], have large k values (e.g., ≥ 100). It would be hard for these codes to adopt the idea of Nos because Nos requires $\Theta(k^2)$ failure domains to function correctly, which is prohibitive when k is large.

Fortunately, Nos and wide-stripe erasure codes target distinct scenarios. Nos optimizes common-case I/Os for fast storage systems with frequent reads and updates, where values of k are moderate. In contrast, wide-stripe erasure codes aim at ultra-low storage consumption and recovery cost for cold storage systems with few reads or updates.

Further, modern datacenters can have tens of thousands of independent servers in a cluster [1, 48]. They can deploy Nos with reasonably large k values if they want. Still, this can be a challenging task as a large cluster face a lot more failure modalities than a small one. To keep the cluster small, one

may, for example, place two failure domains on each node, which halves the cluster size but doubles p.

Extra network bandwidth consumption. Nos consumes extra network bandwidth because of two intrinsic problems.

- First, in each write, Nos replicates data one more time than the Reed-Solomon code. Since the replicas can be transmitted in parallel, it has a limited impact on latencies. Future work may mitigate this problem by multicast to reduce outbound bandwidth consumption.
- Second, Nos needs to read $O(kp)$ objects to recover a desired object when direct recovery is impossible. Though there are no workarounds for this problem, as shown in §4.1, we can reconstruct other lost objects during the recovery. The amortized recovery cost (i.e., number of objects to read) of a lost object is still no more than k.

Also, fortunately, the extra bandwidth overheads are limited since objects are usually small in real-world workloads.

Rigid placement constraints. The most significant limitation of Nos is that it imposes a rigid data placement policy on the cluster. In a cluster of size v, each node can only send/receive data to/from $\Theta(\sqrt{v})$ nodes, leaving the remaining $\Theta(v)$ nodes unused. Also, since v depends on k, it can be difficult to use two coding schemes with different k values for different data in the same cluster. As a result, Nos is far less flexible than conventional coding schemes like the Reed-Solomon code. This is a consequence of the fact that Nos uses a raw SBIBD as its affinity matrix without modification. We hope future work can solve this problem with more advanced theory designs.

5 NOSTOR

To compare Nos with existing erasure coding schemes, we build NOSTOR, an in-memory key-value store prototype that employs Nos for fault tolerance. NOSTOR is implemented in Rust with around 16K lines of code.

5.1 Overview

NOSTOR adopts a C/S architecture. Servers maintain objects and parities in DRAM-resident hashmaps; for this, we use an existing implementation, DashMap [51], which is a sharded hashmap with each shard protected by a reader-writer lock. Clients and servers communicate via remote procedure calls (RPCs) atop fully-userspace RDMA.

Subclusters. NOSTOR employs a unified Nos policy in the entire storage system, i.e., with the same parameters (k, p) and SBIBD instance. There must be at least $v = k^2 - k + 1$ server nodes. If there are more nodes, NOSTOR will create *subclusters* in the entire cluster, each of size v. Subclusters may overlap with each other on some nodes, but NOSTOR will not encode objects from different subclusters together.

NOSTOR divides the key space by hashing for (1) subclusters and (2) primary nodes within each subcluster. Objects do not have predetermined backup servers; instead, primary servers decide backups for objects upon insertion.

Thread model. NOSTOR runs two types of threads on each server: foreground and background.

Foreground threads are RPC threads that perform reads and writes to primaries and dispatch backup write RPCs to other servers. For backup writes, foreground threads only push them into waiting queues to shorten the critical path, instead of digesting them synchronously. Each server maintains a *replication queue* (denoted as q_i) for each replication source i that buffers backup writes from server i. NOSTOR also maintains an index for items in the replication queues to query the yet-to-be-encoded objects during degraded reads.

Background threads pop items from replication queues and digest them. Its tasks include encoding new parities and updating existing parities. We will elaborate on the concrete jobs of background threads in §5.3.

5.2 Handling Client Requests

Client requests consist of `PUT`s and `GET`s. A client sends an RPC to the object's primary server to perform a request.

`PUT`. NOSTOR regards insertions, updates, and deletions as `PUT`s. It complies with the following contract: `PUT`s are committed (i.e., visible and fault-tolerant) by the time they return. To this end, NOSTOR employs a versioning mechanism:

- Each server locally maintains a monotonically increasing sequence number (SN) and assigns it to write requests. Each server also maintains a committed SN (CSN): all writes with SNs no larger than the CSN are committed.
- Instead of a single object, NOSTOR maintains a version queue for each primary object in the hashmap. Accesses to the queues are protected by reader-writer locks.

The detailed `PUT` procedure is as follows. ① Append to the object's version queue and fetch an SN. ② Compute the *delta* between the new value and its previous version. If this is an insertion/deletion, we regard the previous/new value as zero. ③ Replicate the delta to this object's $(p+1)$ backup servers. If this is an insertion, we first select the backup servers. Here, NOSTOR can run arbitrary selection policies and adapt to dynamic data placement requirements, e.g., avoiding unavailable or slow backup servers. ④ Update the CSN when replication finishes. Then, remove outdated versions of the object.

`GET`. The server first queries its hashmap for the object's version queue. If the queue is empty or the queue head is not yet committed (i.e., SN larger than the CSN), the server reports that the object is not found. Otherwise, it finds the object stored in the queue head and replies to the client.

5.3 Digesting Deltas

Background threads (BGTs) digest object deltas sent from their primary servers from the replication queues. To ensure failure consistency, BGTs only process committed deltas iden-

tified by their primaries' SNs and the CSNs, which are piggybacked in the RPCs. Uncommitted deltas will be pushed back into their corresponding replication queues.

Term definition. For convenience, we define several terms that we will frequently use in our following discussion.

- *Encodee* of a parity is an object encoded into this parity. In Nos, all parities should have exactly k encodees, but Nostor relaxes this constraint.
- *Full parity* is a parity with k encodees.
- *Partial parity* is a parity with fewer than k encodees. Nostor consistently attempts to convert partial parities into full parities to lower its storage overheads.

We elaborate on the jobs of BGTs as follows.

5.3.1 Encoding new objects into parities

Each BGT will repeatedly scan all replication queues and try to collect a delta from each queue to encode them together. Specifically, when it pops a delta from a replication queue, it first checks the hashmap for the delta's key. If no such entry exists, i.e., this is a new object, the delta is recognized as an insertion. The BGT will collect this delta and switch to the next replication queue. When it successfully collects k deltas, one from each queue, it will replace them with their XOR sum, i.e., a full parity. Finally, the BGT inserts the full parity, together with the keys of its encodees, into the hashmap.

Avoiding blocking. In practice, a server can temporarily block on some replication sources with no incoming new objects, which can prevent data in other replication queues from being digested and bloat Nostor's memory usage. To address this problem, BGTs do not wait indefinitely to receive from a replication queue; if they timeout, they will skip that queue and move forward. The timeout must be short to reduce CPU cycles wasted on waiting (10 µs in our implementation).

In this case, a BGT will collect fewer than k deltas when it finishes a round of scan, and the XOR sum will become a partial parity. Partial parities do not affect Nos's correctness since they add no extra dependencies among the objects. However, they must get converted into full parities as soon as possible. To this end, each server maintains a *parity queue*, denoted as $\bar{q}_i$, for each replication source i to record partial parities. $\bar{q}_i$ has the following admission rule: it holds partial parities whose encodees are *not* from primary server i. A partial parity can reside in one of all the valid parity queues.

BGTs will actively try to convert partial parities into full parities. Upon collecting a new object from the replication queue q_i, a BGT will first check whether $\bar{q}_i$ is empty. If not, it will pop a partial parity from $\bar{q}_i$ and encode the delta into it. The resulting parity will be pushed into another valid parity queue $\bar{q}_j$ if it is still a partial parity.

Handling concurrent PUTs. Multiple BGTs can concurrently digest deltas of the same object, of which only one may be recognized as an insertion. To this end, the BGT inserts a placeholder entry for the corresponding key in the hashmap if it recognizes a delta as insertion. Therefore, other BGTs will see this placeholder and recognize their deltas as updates. When the BGT finishes encoding the collected deltas, it will replace the placeholders with pointers to the encoded parity.

5.3.2 Updating existing parities

If a BGT pops a delta from a replication queue and finds a corresponding entry in the hashmap, it will recognize the delta as an update. The entry should (eventually) be a parity encoding the delta's belonging object. It will encode the delta into the parity, i.e., replace it with its XOR sum with the delta.

Handling deletions. Deletions can convert full parities into partial parities. Upon deleting an encodee from a full parity, the BGT inserts its metadata into a valid parity queue.

Will partial parities heap up? As long as objects are evenly distributed (which can be achieved with a good key hash algorithm), Nostor can prevent partial parities from heaping up because it is expected that every node will receive the same number of objects from its each replication source. Skewed update traffic will be non-blockingly digested into existing parities and, thus, is not a problem here.

5.4 Failure Recovery

Nostor needs to recover itself if there are failed servers in a subcluster. This task contains three jobs: ① ensuring failure consistency, ② providing degraded read service during the failure, and ③ repairing failed server nodes.

5.4.1 Failure model

Like previous work [6, 25, 36, 56], Nostor assumes fail-stop failures only. Partial network failures or network partitions will likely prevent Nostor from performing PUTs, but since Nostor has no centralized components, clients can still perform GETs on reachable nodes without worrying about split-brain. Byzantine failures are out of the scope of this paper.

5.4.2 Failure consistency

When a server i fails, Nostor must ensure the consistency of its replication targets. Replication target servers of server i do the following steps to ensure consistency. ① Notify all BGTs to stop digesting deltas in q_i and spawn a dedicated thread to collect the SNs of the deltas in q_i. ② Exchange the collected SN information with other replication targets. Thus, the server can find the largest CSN for server i. ③ Discard all deltas in q_i with SNs larger than the CSN. ④ Notify all BGTs to restart digesting deltas in q_i and wait for q_i to become empty.

Note that in step ②, replication targets of node i exchange all collected SNs, instead of the maximum, to correctly find all committed PUTs. After the steps above, all living replication targets of server i will reach a consistent state.

5.4.3 Degraded I/O

Nostor can serve degraded reads to objects whose primaries have failed as soon as Nostor completes the procedure described in §5.4.2 and reaches a consistent state. For simplicity,

Nostor disallows degraded writes. In other words, if a write request matches one of the following conditions, it will be blocked: ① the primary has failed, or ② it cannot replicate to $(p+1)$ living backup nodes.

To perform a degraded read to object x whose primary is server i (failed), the client sends a read RPC to any living server. The server performs the following steps. ① Send RPCs to all living replication targets of server i, which query their hashmaps for the given key and return the metadata of the parities encoding the corresponding object. If the object exists, at least one backup node should reply with non-empty metadata. ② Find the parity that encodes the fewest failed objects, which is the best parity to recover x. ③ Read the parity and all its encodees except x. This may trigger recursive degraded reads, but the recursion will not go deeper, as proved in §4.1. ④ XOR the parity and all the encodees together to decode x.

5.4.4 Node repair

When a new server joins the subcluster to replace a failed one, Nostor starts the node repair procedure. It becomes online and starts serving client requests only after it finishes the following two recovery tasks.

Recover primaries. The server contacts all its replication targets to get a complete list of objects to store a primary copy of. Then, it sends degraded reads requests to them to recover the objects and store the results. Since decoding is performed by other servers, the recovering server will not suffer from inbound network traffic bottleneck.

Recover parities. The server contacts all its replication sources to retrieve all objects that should have been replicated to it. Retrieved objects are pushed into the corresponding replication queues. Then, the server encodes them into parities in the same way as we have discussed in §5.3. There is no need for Nostor to encode objects into the same parities before the failure. The server can perform object retrieval and encoding in parallel to reduce the recovery time.

6 Evaluation

We seek to answer the following questions.

- How does Nostor perform in common cases, and whether it performs well or not, why (§6.2, §6.3)?
- How does Nostor perform in failure recovery (§6.4)?
- How much memory does Nostor use (§6.5)?

6.1 Experiment Setup

We evaluate Nostor on CloudLab [12]. The testbed consists of 16 `c6525-100g` nodes. Every node has the following hardware and software configuration:

- **CPU**: 1× 24-core AMD 7402P, running at 2.80 GHz;
- **RAM**: 128 GB (8× 16 GB 3200 MT/s);
- **NIC**: Mellanox ConnectX-5 Ex 100 Gb (one port);
- **OS**: Ubuntu 22.04 LTS, kernel version 5.19;
- **NIC Driver**: DOCA-OFED 2.10.0;
- **Coding Library**: Intel ISA-L 2.31.1 [19] (AVX2 enabled).

We run two servers on each node so that we can evaluate Nostor with up to $k = 6$ ($v = k^2 - k + 1 = 31$). Note that in deployment, each server should run on a separate failure domain; our setting is due to the lack of physical servers.

Baseline systems. We evaluate Nostor mainly against:

- `Cocytus`: An *inter-object* erasure-coded in-memory object store based on Memcached and TCP/IP network [6]. It uses the Reed-Solomon code. For fairness in comparison, we identified the main logic in its source code and ported it line-by-line into our evaluation framework. `Cocytus` has a similar write path with Nostor: compute the deltas, replicate them, and commit asynchronously. It differs from Nostor in encoding and stripe allocation.
- `PQ`: An *inter-object* object store similar to `Cocytus`, except that it switches from the Reed-Solomon code to the P+Q code in RAID-6 [33]. It uses XOR instead of matrix computation to encode data.
- `Split`: An *intra-object* scheme that splits each object into smaller chunks and encode them into a stripe. It represents Ceph [49, 50], EC-Cache [36], and Hydra [25].
- `Repl`: Ignores k and uses $(p+1)$-way replication for fault tolerance. It offers high performance at the cost of high memory consumption.

We evaluate the systems under three configurations: $(k, p) = (4,2)$, $(6,2)$, and $(6,3)$. Note that `PQ` does not work with $(6,3)$ since it only supports $p = 2$. All baselines use static data placement policies; we will explain this choice in §6.2.4.

Thread configuration. With hyperthreading enabled, we have $24 \times 2 = 48$ cores on each node. The two servers each use 18 cores, with 12 foreground and 6 background threads. `Repl` and `Split` do not need background threads to digest object deltas, so they use all 18 cores to run foreground threads. We run a client on each node that uses the remaining 12 cores, with 32 coroutines multiplexing each thread.

6.2 Microbenchmarks

We evaluate Nostor with the following microbenchmarks:

- 100%-`GET` and 100%-`PUT` with fixed value sizes;
- YCSB [10] A, B, and D (C omitted since it is 100%-`GET`).

The size of the key space is 50 million. Objects are accessed following a Zipfian-0.99 distribution.

6.2.1 100%-GET & 100%-PUT

Figure 3 shows the throughputs. For `GET`, all evaluated systems except `Split` perform similarly well because they only query the hashmap once per `GET`. With value sizes ≤ 256 B, Nostor's `GET` throughputs are 3.92× and 6.06× of `Split` for $k = 4$ and $k = 6$, respectively. These ratios are very close to k, implying that the network IOPS is the bottleneck when

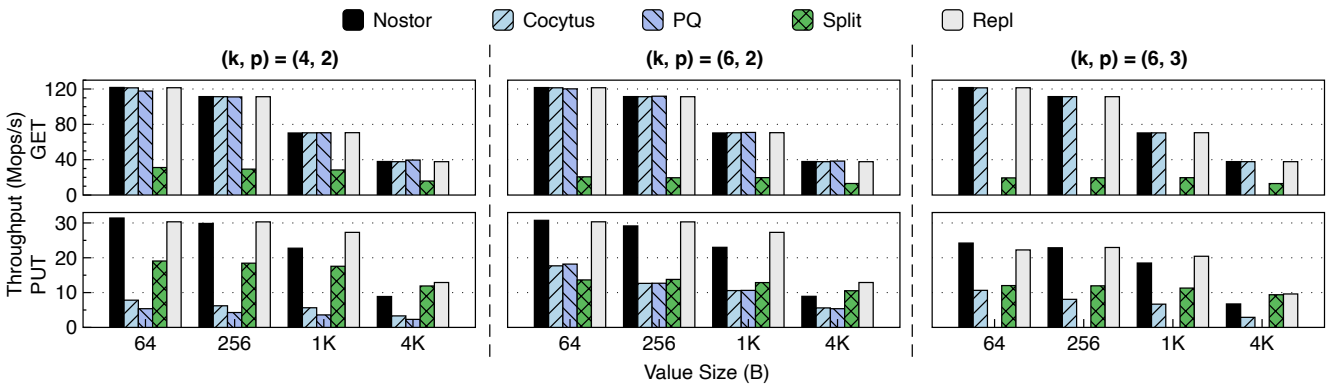


Figure 3: Performance of 100%-GET and 100%-PUT microbenchmarks.

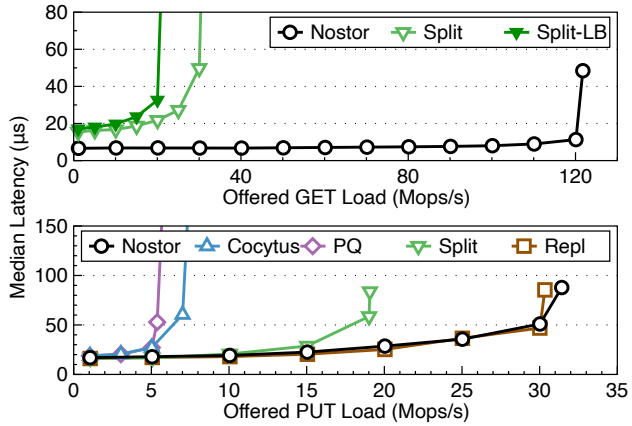


Figure 4: Latency-throughput curves under 100%-GET and 100%-PUT microbenchmarks.

objects are small and that we have pushed the systems to their maximum throughputs. For 1 KB and 4 KB values, the performance bottleneck shifts to the network bandwidth. Still, Nostor delivers 2.40× to 3.57× throughput of `Split`.

For PUT, as expected, `Repl` shows the overall best performance. Nostor performs similarly well for 64 B values; for 1 KB and 4 KB values, it delivers 87.4% and 69.7% throughputs to `Repl`, respectively. This is because the overhead of the one extra replica per PUT starts to manifest as value sizes increase. Also, due to the lowest network bandwidth consumption, `Split` outperforms Nostor for 4 KB values. `Cocytus` and PQ perform the worst (37.3% to 56.6% lower than Nostor on average); we find that it is the synchronized space allocation that hampers their performance the most. They allocate objects in one large stripe using the first-fit policy, serializing all PUTs and severely harming their performance. In Nostor, instead, objects are independent, and such synchronization overhead does not exist in the first place. Overall, Nostor achieves relatively good PUT performances.

Figure 4 shows latency-throughput curves with $(k, p) = (4, 2)$ and 64 B values. For GET, Nostor's curve also represents `Cocytus`, PQ, and `Repl`, which all deliver the best performance. `Split` has higher latencies and lower throughputs due to its high I/O fanout. For PUT, Nostor also has the lowest latencies; interestingly, it even slightly outperforms `Repl`. The reason is that Nostor stores fewer parities than `Repl`'s replicas, resulting in reduced pressure on the hashmap index.

We also make some additional comments to the baselines:

- For `Split`, previous work proposed late-binding, a GET latency optimization technique: fetch one more chunk in the stripe and use the first k that arrive to decode the original data. However, late-binding is unfavorable here because network is always the performance bottleneck, and late-binding can only aggravate it. With $(k, p) = (4, 2)$, `Split`'s throughputs decrease by 15.6% to 28.4% for different value sizes with late-binding. We draw a curve for `Split` with late-binding in Figure 4 (`Split-LB`) which shows that there is no improvement in median latency. Worse, the 99% tail latency even increases by more than 2.2×.
- For `Cocytus`, our implementation delivers 78.5% and 17.9% lower latencies for GETs and PUTs than the original implementation, respectively, with at least a 9.5× throughput improvement, even if we assume the original implementation has 100% scalability. Therefore, we believe that our implementation and evaluations of `Cocytus` are fair.

6.2.2 YCSB

Figure 5 shows the performance of YCSB workloads, where we use 1 KB values as in its original specification. For the write-intensive YCSB-A, Nostor performs similarly to `Repl` and significantly better than `Cocytus`, PQ, and `Split` due to its simplified write path. For the read-intensive YCSB-B and YCSB-D, all systems except `Split` have good performances. These results are all consistent with our conclusions drawn in §6.2.1. Also, overall, all evaluated systems deliver higher throughputs on the YCSB-D workload than on YCSB-B. This is because YCSB-D reads recently inserted objects, which should improve cache efficiency at the servers.

6.2.3 Sensitivity analysis

Figure 6 shows how the performances of the evaluated systems vary with different (k, p) parameters. We run YCSB-A workload under each configuration. Note that we cannot eval-

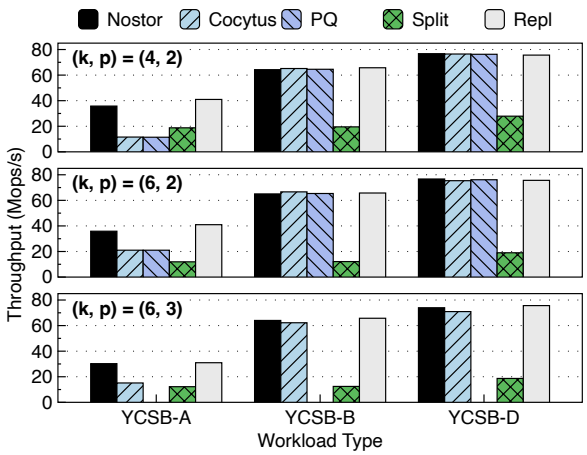


Figure 5: Performance of YCSB workloads.

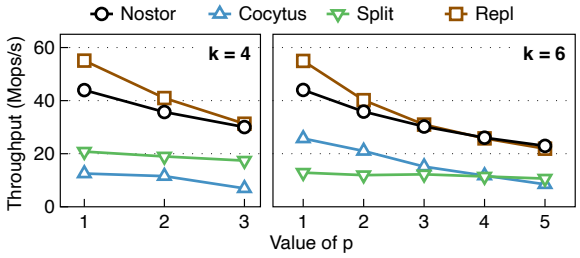


Figure 6: Performance with different (k, p) configurations.

uate PQ here since it only works with $p = 2$.

We make the following two observations. First, Nostor's performance highly depends on p, but k has little impact on Nostor's performance. This is due to Nos's design: its write critical path is similar to primary-backup replication and unaware of k. Second, as p becomes large, the performance gap between Nostor and `Repl` closes. This is because the impact of the extra replica of Nostor has diminishing marginal effects as the total number of replications increases.

6.2.4 How about dynamic data placement?

One may wonder why we only evaluate static data placement policies. To answer this question, we implemented a baseline that adds a dummy MDS to the I/O critical paths to simulate dynamic data placement policies. The MDS just adds a centralized point in the I/O critical paths and does nothing more. It must replace the position of a server (since there are no more cores), so we evaluated all other systems with one fewer server to compare them with it. Figure 7 shows the results. Bottlenecked by the MDS, the baseline delivers the lowest throughputs among the evaluated systems: 89.2% and 72.4% less than Nostor for GETs and PUTs, respectively. This convinces us that we must avoid the centralized MDS and opt for static data placement policies to achieve high performance.

6.2.5 Adaption to runtime data placement constraints

We evaluate how Nostor and other systems can adapt to dynamic data placement constraints at runtime. We run the

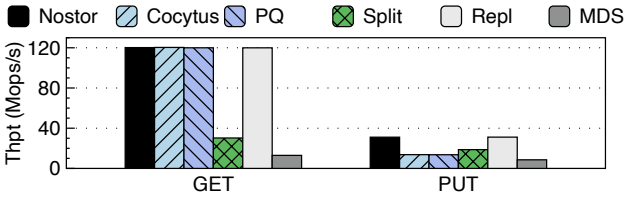


Figure 7: Performance comparison with a baseline system that has an MDS on its I/O critical paths.

YCSB-D workload under $(k, p) = (6, 2)$ and simulate a temporary node slowdown scenario, during which two nodes incur 1 ms extra latency for PUTs (by sleeping). The node slowdown information is broadcasted to every server. Figure 8 shows the results, in which the slowdown starts at $t = 10$ s and ends at $t = 35$ s. `Cocytus` also represents PQ since they are similar.

Nostor can avoid replicating objects to slow nodes in PUTs, so it only suffers from millisecond-level tail latencies caused by objects whose primary nodes are slowed down. In contrast, `Cocytus` suffers from skyrocketed tail latencies (48.2×) because it cannot dynamically place its stripe or circumvent the slow nodes. Due to the read-recently-inserted access pattern of the YCSB-D workload, `Cocytus`'s slow nodes also severely affect its `GET` operations and result in a significant throughput drop of 98.9%. `Repl` suffers from 25.0% lower throughput and 47.7× higher tail latencies due to similar reasons. `Split` is special here because it already suffers from high, millisecond-level tail latencies due to its high network IOPS pressure. The presence of the slow nodes decreases the pressure, causing no negative effects to the tail latencies. However, we observe significantly higher (10.8×) PUT median latencies. This is because `Split`'s PUT operation has a higher fanout than all other evaluated systems; thus, more PUTs are affected by the slow nodes.

Despite the results above, we find that it is possible for `Cocytus`, `Split`, and `Repl` to avoid replicating to slow nodes and, thus, reduce their performance penalties. Specifically,

- `Cocytus` and `Split` can allow dynamic placement of parity chunks in each stripe. For example, when $p = 2$, they can choose 2 nodes out of $(2 + s)$ candidates to place parity chunks, which allows us to tolerate s slow nodes. Here, s is the "degree-of-freedom" of node choices. We use $s = k - (p + 1) = 3$ in this experiment, the same as Nos.
- `Repl` can adopt strategies like hinted handoff [11] to avoid replicating data to slow nodes. Specifically, if an original backup node is slow, the primary node can hand off the object replica to another backup node. The new backup node monitors the state of the original backup node, and hand back the object replica when it becomes normal.

We implement variants of `Cocytus`, `Split`, and `Repl` with features above enabled (denoted as `Cocytus+`, `Split+`, and `Repl+`, respectively) to compare Nostor against them. They are plotted as dotted lines in Figure 8. During the slowdown event, the tail latencies of `Cocytus+` and `Repl+` are similar

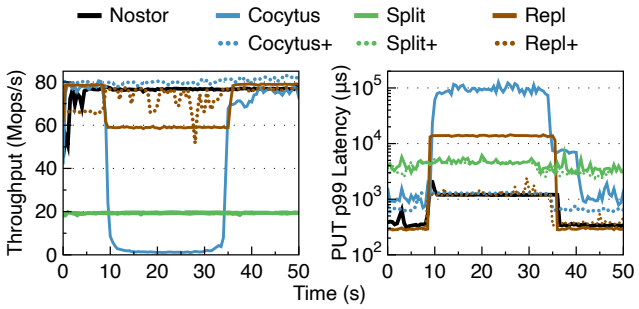


Figure 8: Performance with runtime data placement constraints (two nodes slowed down). *Latency is log-scale.*

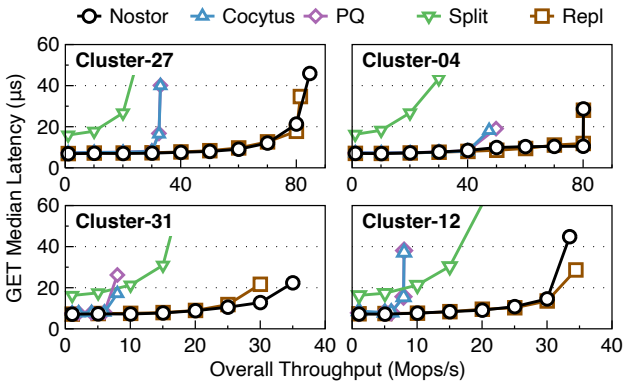


Figure 9: GET performance of Twemcache workloads.

to Nostor. Split+ performs nearly the same with its original version in the figure, while its PUT median latencies now remain stable. Nevertheless, these methods also incur other significant overheads. Specifically,

- Cocytus+'s memory consumption markably increases (16.2% to 62.6% with different object sizes) compared with its original version. This is because allowing dynamic candidates is equivalent to splitting each stripe in Cocytus splits into multiple new stripes with the same primary nodes but different backup nodes. With many more stripes, our worry in §2.2.2 about the static assignment policy becomes true: a stripe might, with a high chance, contain empty chunks and result in a significant memory waste.
- Split+ also consumes more memory because, to avoid aggravating the request fan-out problem, it need to store parity chunks' positions along with the object splits, so that it can retrieve them quickly during degraded reads. This leads to 6.1% to 78.4% extra memory usage.
- Repl+ suffers from two problems. First, its throughput is unstable during the slowdown event because hinted handoff causes load imbalance among normal nodes. Second, after the slow nodes recover, its tail latencies are 28.9% higher than Repl on average since the object hand-back traffic interferes with the client request traffic.

Overall, we observe that it is difficult to perfectly augment these baselines to gracefully tolerate slow nodes. In contrast, Nostor can achieve this goal with little extra efforts.

As a final comment, "broadcasting node slowdown to every server" is only for the convenience of this experiment. In practice, slowdown detection can work in a decentralized manner: each server monitors the performance of object delta replication RPCs and dynamically identifies slow nodes.

6.3 Real-World Trace Benchmarks

We evaluate Nostor with Twitter's Twemcache trace [54] with $(k, p) = (4, 2)$. We select four representative traces covering small/large objects and read/write-intensive workloads; see Table 1. Figures 9 and 10 show the median latency-throughput curves for GET and PUT operations, respectively.

Nostor consistently outperforms other erasure-coded systems with 1.61× to 2.60× throughput improvements. It also slightly outperforms Repl on Cluster-31 for the same reason described in §6.2.1. Nostor and Repl's median latencies are not significantly affected by object sizes: despite the two orders of magnitude difference in average sizes, most objects still fit in one MTU (9000 B in our testbed) and therefore have limited impact on networking overheads.

We observe that all systems except Split perform similarly when the load is low. As the load increases, Cocytus and PQ start to meet throughput bounds due to their poor PUT performance. This also explains why Cocytus and PQ deliver lower throughputs on Cluster-27 than Cluster-12, even if the former has much smaller objects: the higher write ratio significantly hampers their overall performance.

Due to space limits and the fact that Nostor does not specially optimize tail latencies, we omit the tail latency-throughput curve figures. Here, we present some important details about tail latencies. First, Nostor's PUT p99 latencies are superior to all baselines except Repl. On Cluster-31, Nostor outperforms Cocytus/PQ by 33.9% and Split by 6.4%; it underperforms Repl by 26.8%. On Cluster-12, Nostor outperforms Cocytus/PQ by 77.1% and Split by 23.4%; it underperforms Repl by 39.6%. Second, Nostor's GET p99 latencies are similar to Cocytus/PQ/Repl and 68.5% lower than Split. Third, in contrast to the median latencies, the p99 latencies are visibly affected by object sizes. From small objects (Cluster-27/31) to large objects (Cluster-4/12), Nostor's GET/PUT p99 latencies increase by 3.69× and 1.35× on average, respectively. This is because the object sizes shown in Table 1 are the average sizes, while p99 latencies are dominated objects larger than the average. Also, these large objects are more affected by temporary request bursts.

To sum up, Nostor is superior to all evaluated erasure-coded systems in both latency and throughput. It also delivers similar median latencies to Repl, but its tail latencies can sometimes be higher.

	Small objects	Large objects
Read-intensive	Cluster-27 (15% write, object size 74 B, $\alpha = 1.07$)	Cluster-04 (7% write, object size 2506 B, $\alpha = 1.10$)
Write-intensive	Cluster-31 (94% write, object size 56 B, $\alpha = 0.00$)	Cluster-12 (80% write, object size 1074 B, $\alpha = 0.30$)

Table 1: Summary of Twemcache traces used in evaluation. Object size = key + value (averaged). α is the Zipfian parameter.

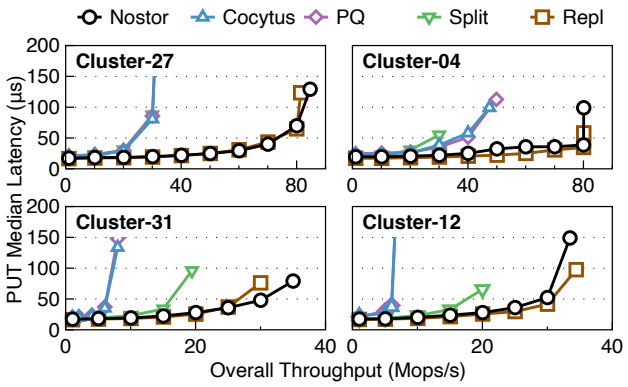


Figure 10: PUT performance of Twemcache workloads.

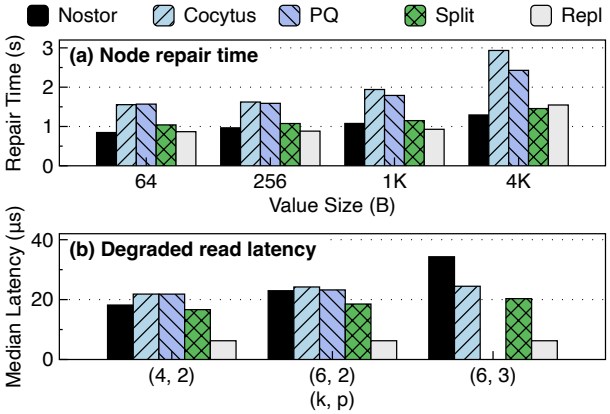


Figure 11: Recovery performance.

6.4 Recovery Performance

6.4.1 Node repair time

We make two servers fail under $(k, p) = (4, 2)$ and then restart them. Each server recovers itself as described in §5.4. Figure 11(a) shows the average time to recover a node. Other (k, p) configurations yield similar trends and are omitted.

Nostor and `Repl` have the highest node repair speed. `Repl` is fast because it does not need to decode any data; instead, it only reads object replicas from backup nodes. Nostor needs to perform decoding, but it has a smaller number of parities ($(p+1)/k\times$ object count) than `Repl`'s replicas ($p\times$ object count). This results in reduced performance overheads caused by inserting into the hashmaps, so Nostor is also fast.

For `Split`, `Cocytus`, and `PQ`, the amount of data to recover is the same. `Split` is slightly (16.4%) slower than Nostor because its data are split into finer-grained chunks, incurring more metadata-related overheads. `Cocytus` and `PQ` are more (88.2%) slower than Nostor because they naturally have low PUT performances. Also, `Cocytus` recovers (9.1%) slower than `PQ` with value sizes ≥ 256 B. We attribute this to their coding scheme difference: `Cocytus` decodes data with matrix multiplication, while `PQ` uses XOR. ISA-L performs XOR and matrix multiplication similarly fast on 64 B data, but when data is larger, XOR becomes 4.06× faster than matrix multiplication. This reduces the decode time of `PQ`.

6.4.2 Degraded read latency

We measure 64 B degraded read latency under each (k, p) configuration with p failed nodes. Figure 11(b) shows the results. `Repl` is the fastest since it only needs to read a replica. `Split` is the second fastest since the amount of data it reads is only $1/k$ to other erasure-coded systems. For $(4, 2)$ and $(6, 2)$ configurations, Nostor's degraded read latency is 16.5% higher than `Split` on average. However, it is also 11.0% faster than `Cocytus` and `PQ` because it only uses XOR to decode data (`PQ` also performs bit-shift and bitwise-AND).

For Nostor with $(k, p) = (6, 3)$, we carefully choose the failed nodes such that the worst case, i.e., recursive degraded read, will be triggered. In this case, due to an extra round of recovery, Nostor's degraded read latency rises to the highest, 35.0% higher than `Cocytus` and 62.4% higher than `Split`. Since such degraded reads should be rare among all read operations, we believe the increased latency is acceptable.

6.5 Memory Consumption

We populate the systems with 50 million objects and measure their memory consumption. Figure 12 shows the results, in which Nostor's memory usage (in GBs) is labeled above the bars. `Cocytus` also represents `PQ` since they are similar.

We make the following observations. First, Nostor incurs extra metadata overheads because it must store the encodees' keys along with each parity, which explains why it consumes more memory from $(k, p) = (4, 2)$ to $(6, 2)$. As the value size increases, the overheads become negligible. Second, `Cocytus`'s memory usage is the lowest with 64 B values but rises as the value size increases, showing that its memory allocator produces "holes" in the stripes and wastes memory: as value sizes increase, metadata overheads become negligible, and the wastes start to manifest. Third, `Repl` consumes 1.23× to 2.35× memory compared with Nostor. With the performance similarity of Nostor and `Repl` in mind, we believe Nostor offers a favorable performance-storage tradeoff over replication. Finally, even if we continuously apply a skewed (Zipfian-0.99) write-intensive workload to Nostor, its memory usage still keeps stable. This shows that Nostor can digest object deltas fast enough and avoid bloating its memory usage.

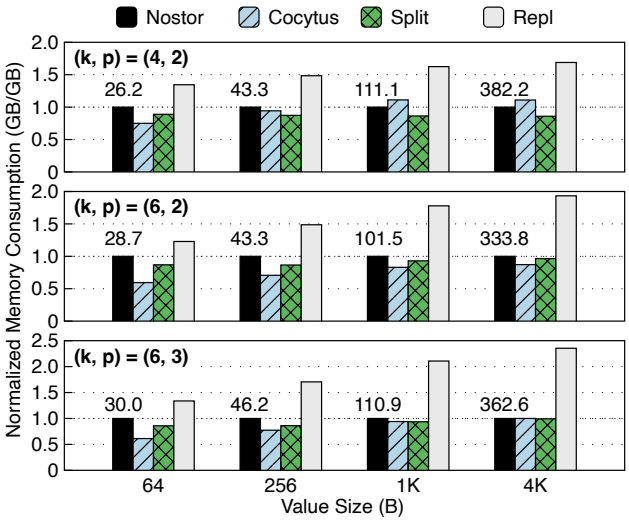


Figure 12: Normalized memory consumption.

In §6.2.5 we discussed slowdown-tolerant variants of the baselines, namely `Cocytus+`, `Repl+`, and `Split+`. Here, we discuss a bit more about their memory consumption. Compared to Nostor, `Cocytus+` consumes 30.9% less to 35.1% more memory. Its relative memory usage increases as the object size grows, consistent to our observation of `Cocytus` above. `Split+` consumes 9.4% less to 57.4% more memory. In contrast to `Cocytus+`, its relative memory usage decreases as the object size grows. The reason is that the amount of memory to store parity chunks' positions are constant, which becomes less significant when objects grows larger.

7 Related Work

Block designs and codes. Block design is a combinatorics concept with a long history dating back to the mid-19th century; the SBIBD used in this paper is a special kind of block design. In the 1950s, it became clear that block designs are strongly related to erasure coding and error-correcting codes (ECCs) [3, 35, 38]. For example, $(7,3,1)$-SBIBD corresponds to the well-known 7-bit Hamming code with 4 data bits and 3 parity bits. Other block designs can also be used to design codes like the Low-Density Parity-Check (LDPC) code [2].

While block designs have been used in the design of coding schemes since decades ago, their usage is still limited to conventional stripe-based codes. These coding schemes are unfit for new high-speed storage systems. To our knowledge, Nos is the first to use the block design theory to construct an erasure coding scheme without stripes.

Erasure coding for in-memory storage systems. There have been many efforts to apply erasure coding to in-memory storage systems. Some are similar to Nostor in that they ensure all returned writes are fault-tolerant. Cocytus [6] pre-allocates large empty stripes and allocates objects of different sizes within the chunks, which is shown to be inefficient. RE-Store [28] combines replication and erasure coding by replicating each object into two copies to reduce one parity. Though faster in recovery, this approach suffers from both replication's high memory consumption and erasure coding's computation overheads. EC-Cache [36] encodes each object into a stripe by splitting them into chunks. It mainly targets load balance in a system atop the relatively slow TCP/IP network, but falls short in performance-critical scenarios atop the fast RDMA network.

Other systems do not offer the contract that returned writes are committed. BCStore [27] buffers writes at clients until there is enough data to form a stripe, risking the loss of buffered data. Hydra [25] and Carbink [56] assume the far memory model, i.e., only one process running on a single node can access the storage system. Therefore, they do not need to encode data on the same node as the process. However, all data is immediately lost if that node or process fails.

Different encode methods. Erasure coding schemes mainly fall into two categories according to their methods of encoding data: matrix multiplication [17, 18, 20, 29, 37, 46] and XOR [14, 33, 34, 38]. In §6, we have shown with `Cocytus` and `PQ` that the encode method does not greatly affect the performance; instead, the write path design does.

From replication to erasure coding. Nos consists of a synchronous replication phase and an asynchronous coding phase. Many existing systems adopt similar ideas, including f4 [31], HDFS [43], HACFS [52], and RCStor [39]. In these systems, objects are first stored in a replicated storage pool and then moved to an erasure-coded pool in the background. Nos's approach differs from theirs in that it only has one storage pool and encodes object replicas directly into parities. This avoids the movement of the objects' primary copies and reduces the complexity of the entire storage system.

8 Conclusion

This paper proposes Nos, a stripeless data placement scheme for erasure-coded storage systems. It enables failure recovery by selecting backup nodes for objects according to a combinatoric structure called SBIBD. Atop of Nos, we build Nostor, a fault-tolerant distributed in-memory key-value store, to demonstrate the effectiveness of Nos. Evaluations show that Nostor outperforms existing erasure-coded systems on throughput and write latency.

Acknowledgment

We sincerely thank our shepherd for helping us improve the paper. We are also grateful to all reviewers of this paper for their helpful comments and feedback. This work is supported by the National Natural Science Foundation of China (Grant No. U22B2023). This work would not have been possible without the guidance of Prof. Yuchun Ma, who teaches Combinatorics at Tsinghua University with great dedication.

A Artifact Appendix

Abstract

The artifact consists of the Rust code of Nostor and baseline systems used in evaluation, runner scripts to validate their functionalities and reproduce the experiments, and related description files. It is intended for validating the claims made in the paper and facilitating further research on stripeless data placement in erasure-coded storage.

Scope

At a high level, the artifact allows its users to validate the following claims in the paper:

- Nostor outperforms stripe-based erasure-coded storage systems in terms of common-case performance.
- Nostor performs similarly with `Repl` (replication) with significantly less memory consumption.
- Nostor can recover from client failures.

These claims correspond to the questions raised at the start of the Evaluation section (§6).

The main purpose of the artifact is to validate the claims above. Aside from that, since the artifact contains full implementations of Nostor and baselines, it can be used as a starting point for further research on erasure coding and distributed in-memory storage systems.

Contents

The artifact contains source code, scripts, and other necessary components. Below, we explain the contents aside from the README. More detailed descriptions can be found in `HOWTOUSE.md` in the artifact repository.

Source code. The `nos` directory contains the source code of Nostor and baseline system servers. The `nos-cli` directory contains the source code of the client. Nostor's implementation is in `nos/src/objstore`, while the baseline systems are in `nos/src/bin`. Experiment infrastructure is in `nos-cli/src/runners`.

Scripts. The `scripts` directory contains runner scripts for running experiments and plotter scripts for visualizing the results. Scripts for validating the functionalities (i.e., AE scripts) are in the `ae-functional` subdirectory. Other scripts serve as building blocks of the AE scripts, such as running a single experiment. The configuration files used by the scripts are in the `config` directory.

Other components. The `trace-preprocess` directory contains the preprocessor for Twitter's Twemcache trace. For the real trace files, please refer to Twitter's original repository. After being processed, the traces can be consumed and replayed by the clients. The `rrppcc` directory contains the RPC engine implementation. The `data` directory contains the original data. Other directories and files are miscellaneous components, e.g., C library bindings and documentation.

Hosting

Nostor's artifact repository is hosted on GitHub at https://github.com/IcicleF/Nos on the `master` branch. The first usable commit version is the initial commit (`f6f9741`). However, due to possible future bugfixes and updates, please always use the latest commit.

Requirements

To reproduce the experiments, the testbed must have:

- At least 16 Linux servers with Internet access
- One server has password-less SSH access to all servers
- RDMA network + Mellanox ConnectX-5 or newer RNICs
- CPUs with AVX-2 support
- DOCA-OFED, which can be downloaded here
- Rust 1.86 or newer
- Intel ISA-L 2.31.1 or newer

Fewer servers or less advanced RNICs can be used if the user only wants to verify Nostor's functionalities. For detailed instructions, please refer to the README file in the artifact repository.

References

[1] Alibaba Cloud MaxCompute. The Big Data Platform Behind Alibaba's E-Commerce Systems - Alibaba Cloud Community. https://www.alibabacloud.com/blog/the-big-data-platform-behind-alibabas-e-commerce-systems_595931, March 2020.

[2] B. Ammar, B. Honary, Y. Kou, and S. Lin. Construction of low density parity check codes: a combinatoric design approach. In *Proceedings of the IEEE International Symposium on Information Theory*, page 311. IEEE, June 2002. https://ieeexplore.ieee.org/document/1023583.

[3] E. F. Assmus and J. D. Key. *Designs and their Codes*. Cambridge University Press, 1 edition, August 1992. https://www.cambridge.org/core/product/identifier/9781316529836/type/book.

[4] Richard A. Brauldi. *Introductory Combinatorics*. Pearson Education, 5 edition, 2010.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020. https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[6] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and Available In-Memory KV-Store with Hybrid

Erasure Coding and Replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 167–180, Santa Clara CA USA, June 2016. USENIX. https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng.

[7] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, pages 239–252, Santa Clara CA USA, February 2020. USENIX. https://www.usenix.org/conference/fast20/presentation/chen-jiqiang.

[8] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 539–551, 2017. https://www.usenix.org/conference/atc17/technical-sessions/presentation/chen-yu-lin.

[9] Liangfeng Cheng, Yuchong Hu, Zhaokang Ke, Jia Xu, Qiaori Yao, Dan Feng, Weichun Wang, and Wei Chen. LogECMem: coupling erasure-coded in-memory key-value stores with parity logging. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, pages 1–15, St. Louis Missouri, November 2021. ACM. https://dl.acm.org/doi/10.1145/3458817.3480852.

[10] Brian Cooper. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, February 2022.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, Stevenson Washington USA, October 2007. ACM. https://dl.acm.org/doi/10.1145/1294261.1294281.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 1–14, Renton WA USA, July 2019. USENIX. https://www.usenix.org/conference/atc19/presentation/duplyakin.

[13] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, August 2004. https://www.linuxjournal.com/article/7451.

[14] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, January 1962.

[15] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with InfiniSwap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 649–667, Boston MA USA, March 2017. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[16] David Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, April 2021.

[17] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P C Lee, Weichun Wang, and Wei Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, pages 233–248. USENIX, February 2021. https://www.usenix.org/conference/fast21/presentation/hu.

[18] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 15–26, Boston MA USA, June 2012. USENIX. https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang.

[19] Intel Corporation. Optimizing Storage Solutions Using the Intel® Intelligent Storage Acceleration Library. https://www.intel.com/content/www/us/en/developer/articles/technical/optimizing-storage-solutions-using-the-intel-intelligent-storage-acceleration-library.html, September 2014.

[20] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *Proceedings of the 21st USENIX Conference on File and*

Storage Technologies (FAST '23), pages 1–16, Santa Clara CA USA, February 2023. USENIX. https://www.usenix.org/conference/fast23/presentation/kadekodi.

[21] Thomas P Kirkman. On a problem in combinations. *Cambridge and Dublin Mathematical Journal*, 2:191–204, 1847.

[22] Thomas P Kirkman. On the perfect r-partitions of r^2-r+1. *Transactions of the Historical Society of Lancashire and Cheshire*, 9:127–142, 1857. https://www.hslc.org.uk/wp-content/uploads/2017/10/9-12-Kirkman.pdf.

[23] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST '15)*, pages 1–14. IEEE, May 2015. https://ieeexplore.ieee.org/document/7208288.

[24] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, pages 553–569, New York, NY, USA, October 2021. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3477132.3483546.

[25] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. Hydra: Resilient and Highly Available Remote Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, pages 181–197, Santa Clara, CA, February 2022. https://www.usenix.org/conference/fast22/presentation/lee.

[26] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 137–152, Shanghai China, October 2017. ACM. https://dl.acm.org/doi/10.1145/3132747.3132756.

[27] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. BCStore: Bandwidth-Efficient In-memory KV-Store with Batch Coding. In *IEEE 33rd International Conference on Massive Storage Systems and Technology (MSST '17)*, pages 1–13, Santa Clara CA USA, May 2017. IEEE. https://storageconference.us/2017/Papers/BCStore-BandwidthEfficientKV-Store.pdf.

[28] Yuzhe Li, Jiang Zhou, Weiping Wang, and Yong Chen. RE-Store: Reliable and Efficient KV-Store with Erasure Coding and Replication. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, September 2019. https://ieeexplore.ieee.org/document/8891013.

[29] Qing Liu, Dan Feng, Hong Jiang, Yuchong Hu, and Tianfeng Jiao. Systematic Erasure Codes with Optimal Repair Bandwidth and Storage. *ACM Transactions on Storage*, 13(3):26:1–26:27, September 2017. https://dl.acm.org/doi/10.1145/3109479.

[30] Youyou Lu, Jiwu Shu, Tao Li, and Youmin Chen. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[31] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 383–398, Broomfield CO USA, October 2014. USENIX. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar.

[32] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, August 2013. https://dl.acm.org/doi/10.14778/2536222.2536234.

[33] James S. Plank. The RAID-6 Liberation Codes. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, pages 97–110, San Jose CA USA, 2008. USENIX. https://www.usenix.org/conference/fast-08/raid-6-liberation-codes.

[34] James S. Plank. Erasure Codes for Storage Systems: A Brief Primer. *;login: USENIX Mag.*, 38(6):44–50, December 2013. https://www.usenix.org/system/files/login/articles/10_plank-online.pdf.

[35] Eugene Prange. The use of coset equivalence in the analysis and decoding of group codes. https://apps.dtic.mil/sti/pdfs/AD0226767.pdf, June 1959.

[36] K. Vinayak Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah GA USA, November 2016. USENIX. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/rashmi.

[37] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960. http://epubs.siam.org/doi/10.1137/0108018.

[38] L. Rudolph. A Class of Majority Logic Decodable Codes. *IEEE Transactions on Information Theory*, 13(2):305–307, April 1967. https://ieeexplore.ieee.org/document/1053994.

[39] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. Geometric Partitioning: Explore the Boundary of Optimal Erasure Code Repair. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, pages 457–471, Virtual Event Germany, October 2021. ACM. https://dl.acm.org/doi/10.1145/3477132.3483558.

[40] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 69–87, Carlsbad CA USA, 2018. USENIX. https://www.usenix.org/conference/osdi18/presentation/shan.

[41] Zhirong Shen and Patrick P. C. Lee. Cross-Rack-Aware Updates in Erasure-Coded Data Centers. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP '18, pages 1–10, New York, NY, USA, August 2018. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3225058.3225065.

[42] Haiyang Shi and Xiaoyi Lu. INEC: Fast and Coherent In-Network Erasure Coding. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*, pages 1–17, Atlanta GA USA, November 2020. IEEE. https://ieeexplore.ieee.org/document/9355252/.

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10. IEEE, May 2010. https://ieeexplore.ieee.org/document/5496972.

[44] James Singer. A Theorem in Finite Projective Geometry and Some Applications to Number Theory. *Transactions of the American Mathematical Society*, 43(3):377–385, 1938. https://www.jstor.org/stable/1990067.

[45] Jacob Steiner. Combinatorische Aufgaben. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1853(45):181–182, 1853. https://www.degruyter.com/document/doi/10.1515/crll.1853.45.181/html.

[46] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. Clay Codes: Moulding MDS Codes to Yield an MSR Code. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, pages 139–154, Santa Clara CA USA, 2018. USENIX. https://www.usenix.org/conference/fast18/presentation/vajha.

[47] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, pages 267–281, Santa Clara CA USA, 2020. USENIX. https://www.usenix.org/conference/fast20/presentation/wang-ao.

[48] Andy Warfield. Building and Operating a Pretty Big Storage System (My Adventures in Amazon S3). In *The 21st USENIX Conference on File and Storage Technologies (FAST '23)*, Santa Clara CA USA, February 2023. USENIX. https://www.usenix.org/conference/fast23/presentation/warfield.

[49] Sage Weil. Ceph.io — New in Luminous: Erasure Coding for RBD and CephFS. https://ceph.io/en/news/blog/2017/new-luminous-erasure-coding-rbd-cephfs/, October 2017.

[50] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320, Seattle WA USA, November 2006. USENIX. https://www.usenix.org/legacy/events/osdi06/tech/weil.html.

[51] Joel Wejdenstål. DashMap. https://github.com/xacrimon/dashmap, November 2023.

[52] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A Pease. A Tale of Two Erasure Codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 213–226, Santa Clara CA USA, February 2015. USENIX. https://www.usenix.org/conference/fast15/technical-sessions/presentation/xia.

[53] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, pages 221–234, Boston MA USA, February 2019. USENIX. https://www.usenix.org/conference/fast19/presentation/yang.

[54] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020. https://www.usenix.org/conference/osdi20/presentation/yang.

[55] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. Erasure Coding for Small Objects in In-Memory KV Storage. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–12, Haifa Israel, May 2017. ACM. https://dl.acm.org/doi/10.1145/3078468.3078470.

[56] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant Far Memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 55–71, Carlsbad CA USA, 2022. USENIX. https://www.usenix.org/conference/osdi22/presentation/zhou-yang.



PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection

Hayley LeBlanc
University of Texas at Austin

Jacob R. Lorch
Microsoft Research

Chris Hawblitzel
Microsoft Research

Cheng Huang
Microsoft

Yiheng Tao
Microsoft

Nickolai Zeldovich
MIT CSAIL and Microsoft Research

Vijay Chidambaram
University of Texas at Austin

Abstract

Storage systems must maintain integrity even after rare and difficult-to-test-for conditions like power losses and media errors. Formal verification presents a promising avenue to ensure storage systems are resilient, but current approaches involve significant complexity and rely on verification constructs or forms of logic beyond what most verifiers natively support. In this paper, we present two new verification techniques that rely only on standard constructs provided by most verification tools such as Hoare logic, ghost variables, and quantifiers. First, we introduce PoWER (Preconditions on Writes Enforcing Recoverability), a novel approach to verifying crash consistency that encodes its requirements in the preconditions of storage API methods. Second, we present a new model of media corruption for provable corruption detection on any type of storage device. To demonstrate the power of these new techniques, we use them to build two verified storage systems using two different verification frameworks. We build and verify the key-value (KV) store CapybaraKV using Verus and the notary server CapybaraNS using Dafny. Both systems are built for persistent memory (PM), which we target due to new challenges it presents to building resilient storage systems. We develop new techniques to address these challenges, including the *corruption-detecting Boolean*, a new primitive for atomic checksum updates. Both systems verify in under a minute, and CapybaraKV achieves performance competitive with similar unverified PM KV stores.

1 Introduction

Storage systems must preserve the integrity of stored data even if rare events like power losses or media errors occur. However, ensuring that systems are crash consistent and can detect corruption is difficult. There has been extensive prior work on testing storage systems for these properties, but testing cannot guarantee the absence of bugs [14, 18–20, 35, 36, 44, 46–48, 50, 52, 69]. Program verification offers a promising alternative: use machine-checked proofs to ensure resiliency and correctness properties [1, 7, 9, 21, 22, 38, 57].

However, verifying that storage systems are crash-consistent and corruption-resistant is challenging for three reasons. First, verifying crash consistency requires specialized tools that exacerbate the learning curve for verification. Second, prior work on provably detecting corruption makes strong assumptions about data layouts that are not compatible with all systems [21]. Third, verified storage systems have not been able to achieve state-of-the-art performance thus far [7–9, 21, 57]. As a result, mainstream adoption of verification for storage systems has been low.

In this paper, we present a novel approach for verifying that storage systems (1) are crash-consistent and (2) can provably detect corruption. Our approach is tool-agnostic: it only relies on Hoare logic [17, 25], ghost variables, and quantifiers, features supported by most verification tools. This enables us to implement our approach in existing verification tools, such as Dafny [45] and Verus [43], without changing the tools themselves. Our model of corruption enables provable corruption detection without imposing restrictions on data placement.

To demonstrate our approach, we build two verified storage systems, a key-value store and a notary service, using two different verification frameworks. We show that the key-value store achieves performance competitive with similar unverified systems.

Verifying crash consistency. A key challenge when verifying crash consistency is reasoning about *intermediate system states* that may occur when an operation is interrupted by a crash [9]. This is difficult because Hoare logic [17, 25], the basis of most verification tools, trades the ability to describe these intermediate states for efficient modular verification. In Hoare logic, a function is specified by preconditions that must hold when it is invoked and postconditions that must hold when it completes. Its internal behavior is otherwise unconstrained.

Prior verified storage systems have introduced new frameworks and features, beyond what standard verifiers support, to facilitate reasoning about intermediate states. For example, the FSCQ file system extends Hoare logic with *crash conditions*, predicates about intermediate disk states of a function [9], in the Rocq proof assistant [11]. The VeriBetrKV key-value store adds TLA-style [41] reasoning to Dafny [45] to support *state machine refinement* proofs about crash behaviors [21]. These approaches, while powerful, all have a steep learning curve and are tied to certain verification tools.

We introduce PoWER (Preconditions on Writes Enforcing Recoverability), a new approach to reasoning about storage system crashes that, unlike prior work, relies only on standard

constructs provided by most verification tools. The key insight behind PoWER is that new forms of logic or TLA-style reasoning are not required to reason about crashes; we can instead add a precondition to methods that perform durable updates stating that all resulting crash states must be legal. This can be done with Hoare logic and quantifiers, which are supported by most verification tools. In PoWER, when developers call the write API, they are required to supply a proof that satisfies the precondition that the write will always leave the storage system in a crash-consistent state.

The need to write such proofs may seem daunting, but we show how it can be greatly simplified by turning common intuitions used by storage systems developers into a set of proof strategies based on system domain knowledge. These strategies are based on the insight that most crash-consistency proofs can be discharged only by reasoning about the *location* of the write, not its contents or specific crash states. For example, if a write only modifies bytes that are logically unreachable, it is clear that the write will not cause inconsistency.

PoWER does not sacrifice soundness for flexibility. We prove that its guarantees correspond to crash conditions in Crash Hoare logic [9] and to crash invariants in Perennial [6] using machine-checked proofs.

Verifying corruption detection. Many storage systems use cyclic redundancy checks (CRCs) to detect bit corruption due to media errors [4, 16, 21, 66, 67]. As with crash consistency, developers of verified systems must determine the expected model of corruption and CRC properties upon which to base their proofs. The only verified storage system supporting corruption detection, VeriBetrKV [21], assumes that the checksum is embedded with the data, and that both are updated atomically. This is not true of many real-world storage systems [58] (especially on byte-addressable memories), thus limiting its adoption.

We present a new model of data corruption based on the theoretical properties of CRCs and provide libraries and functions for developers writing verified corruption-detecting systems. Our model imposes no restrictions on where data and its checksum are stored and does not require them to be atomically updated together.

Verified systems. To demonstrate PoWER and our corruption model, we build two verified persistent memory (PM) storage systems: CapybaraKV, a key-value store written in Verus [43], and CapybaraNS, a notary service written in Dafny [45]. PoWER and our CRC model are not restricted to PM; we target PM because it presents novel crash-consistency and corruption-detection challenges not encountered with traditional storage.

One such challenge is that PM's fine-grained atomic write size (8 bytes) [64] makes it impossible to atomically update CRCs with data using standard techniques, which were developed for storage devices with much larger access granularity. We develop a new primitive, the *corruption-detecting Boolean* (CDB), to support provably crash-atomic CRC updates. We use CDBs in both CapybaraKV and CapybaraNS and find that they simplify many crash-consistency proofs.

To illustrate that PoWER is compatible with coarse-grained concurrency (if used with a tool like Verus or Perennial that supports concurrency), we extend CapybaraKV to support two forms of concurrency. One extension uses a reader-writer lock to permit concurrent readers or a single writer at a time, and another extension uses sharding to permit concurrent writers to keys falling in different shards.

We focus our evaluation on CapybaraKV and find that it achieves competitive performance with state-of-the-art unverified PM KV stores, outperforming them on many benchmarks. CapybaraKV is designed for an Azure Storage system that uses battery-backed DRAM, and has been integrated with a prototype Rust version of that system.

Our proposed approaches and systems have several limitations. First, PoWER relies on features that are present in most, but not all, verifiers. Highly-automated tools like Yggdrasil [57] and TPot [5] have limited support for quantifiers, which makes them incompatible with PoWER. Second, while PoWER can be used to verify concurrent systems like those discussed above, it does not support arbitrary fine-grained concurrency involving concurrent writes to the same storage region. Finally, as with all verified systems, the correctness of CapybaraKV and CapybaraNS depends on the correctness of their specifications and of the verifiers and compilers themselves.

In summary, this work makes the following contributions:

1. PoWER, a tool-agnostic way to verify crash consistency (§3.1);
2. A set of proof strategies and library functions to simplify crash-consistency proofs based on domain knowledge and natural intuitions about storage systems (§3.3);
3. A flexible and sound model of data corruption that facilitates proofs about the absence of corruption (§4.1);
4. The corruption-detecting Boolean, a useful new primitive for reasoning about CRC updates on persistent memory (§4.2); and
5. CapybaraKV and CapybaraNS, two new verified storage systems in different languages that demonstrate our new methods (§5).

2 Background and related work

In this section, we discuss how existing approaches establish crash consistency and corruption detection to motivate the need for our improved approaches.

2.1 Identification of storage system bugs

Crash-consistency and corruption-detection bugs impact even the most mature storage systems. Prior work on crash-consistency testing tools [36, 48, 52, 69] has found many previously undiscovered bugs in these systems.

Persistent memory (PM). Persistent memory devices provide byte-addressable durable storage at near-DRAM latencies [31, 68]. PM is generally mapped into an application or storage system's address space and accessed via memory load and store instructions [54, 64]. Unlike block devices, PM uses fine-grained atomic writes (8 aligned bytes [64]), and updates must be explicitly flushed from volatile CPU caches to ensure persistence. Recent research on PM storage systems has shown that they are prone to subtle crash-consistency bugs [14, 18–20, 35, 44, 46, 47, 50]. These bugs arise due to both PM's complex low-level interface and higher-level issues in new PM-specific design patterns.

Although PM is not yet widely deployed, cloud services like Azure Storage already use PM in production [40]. A variety of PM-related hardware offerings are currently in development [15, 24, 37, 63] and the CXL.mem protocol is expected to support PM [2].

2.2 Formal verification of crash consistency

Software verification tools are best suited for verifying properties encodable with Hoare logic [17,25], and crash consistency does not seem to fit this mold. In Hoare logic, one writes specifications for functions in the form of preconditions, which must be true when the functions are invoked, and postconditions, which must be true when the functions complete. However, crashes may occur partway through function execution, and Hoare logic does not let a developer directly specify conditions that must hold throughout the body of each function. For this reason, there has been much research proposing specialized methodologies for proving crash consistency.

Crash Hoare logic. Crash Hoare logic (CHL) is used by FSCQ [9], an xv6-like file system implemented and verified in Rocq [11]. CHL extends traditional Hoare reasoning with *crash conditions* that describe system state in the event of a crash. CHL's support for crash conditions is based on separation logic [53], which enables describing disjoint resources (e.g., different parts of a disk) with separate predicates. CHL is currently only supported by Rocq-based verification tools, which have a steep learning curve. Furthermore, Rocq programs can only be run by first extracting them to Haskell or OCaml, which limits their run-time performance.

CHL requires built-in language support or the ability to perform meta-level reasoning about the language's own semantics. Languages like Verus and Dafny would require a substantial overhaul, including changes to the core language, to support CHL. Indeed, some of us are part of the Verus development team, which has no plans to support CHL because it would require significant effort and would only be used by a small subset of users.

In contrast, PoWER can be used with any verifier that supports standard Hoare logic, ghost variables, and quantifiers, including verifiers like Verus [43] that produce fast code. Furthermore, we believe that PoWER would remain a valuable contribution even in the unlikely event that CHL support were added to Verus, because it could be used with *any* verification tool.

Crash invariants. GoJournal [7] and other systems built on top of Perennial [6] reason about crash safety as a form of concurrency, where a crash causes existing threads to stop executing and a new recovery thread to start executing. In particular, Perennial formalizes crash reasoning using *crash invariants* [6, §5.1], which are built on top of *atomic invariants*. Atomic invariants are properties that are verified to hold at the end of every atomic execution step. Many frameworks for verifying concurrent software (including Iris [33] and Verus) support such atomic invariants, but tools that do not handle concurrency (such as Dafny [45], Prusti [65], and Creusot [13]) lack this support. PoWER can thus be used even with tools that lack support for atomic invariants.

PoWER is also useful with tools that *do* support atomic invariants, but which make it cumbersome to directly state Perennial's logically atomic crash specifications [7, §6.2], such as Verus. Parts of the code requiring concurrency reasoning must use atomic invariants, but other code only needs to reason about PoWER specifications and can thus avoid the complexity of logically atomic crash specifications. (See §3.4 and §5.1.2.)

State machine refinement. State machine refinement was originally developed for verification of distributed systems in the IronFleet project [22]. VeriBetrKV [21] applies the technique to verify a key-value store based on the intuition that storage software interacting with a disk is similar to unreliable nodes interacting in a distributed system. Anvil [59] uses it to verify liveness properties of cluster management controllers. In these systems, each component (e.g., a single node, or a storage system journal) is modeled as a state machine and proven correct in isolation using Hoare logic assuming a synchronous and crash-free environment. Components are then proven correct via a series of state machine refinement proofs using TLA-style reasoning, which can also prove properties related to asynchrony, crash consistency, and liveness. IronFleet and VeriBetrKV are implemented and verified in Dafny with custom-built libraries for TLA-style reasoning.

These proofs require additional infrastructure on top of a TLA library. They are also labor intensive, since developers need to reason about low-level sequences of steps and write refinement proofs. In contrast, our techniques do not require additional infrastructure or separate refinement proofs, and let developers reason about their systems at a higher level.

Push-button verification. Push-button verification [51]

trades expressivity for the ability to verify code without writing any proofs. Crash refinement [57] is a technique for push-button crash-consistency verification in which the developer specifies a (finite) set of all possible crash schedules and requires that all executions obey the specification. This significantly reduces annotation and proof burden but is limited to simple, bounded systems. Other highly-automated verification techniques, such as the approach taken in TPot [5], have not been applied to storage verification.

Summary of comparisons. In contrast to the above, PoWER relies only on standard verification constructs, letting developers use whatever verification tool they feel is best suited for their project. Indeed, based on the recent influx of Rust-based verification tools, we expect that future improvements in verification tooling may come in the form of new languages and verifiers. Techniques that require developers to build complex frameworks or rely on specific language features will be less useful than approaches like PoWER.

2.3 Formal verification of corruption detection

VeriBetrKV [21] formally reasons about the use of checksums for corruption detection. However, its corruption-detection axiom strongly dictates how data must be checksummed and where the checksum must be stored. VeriBetrKV does not allow checksumming data across blocks, and precludes storing the checksum separately from the data. As we discuss in §4.1, this is overly restrictive for the systems we target. In contrast, our proposed model imposes no limitations on what data is checksummed or on where the checksum and data are stored. Moreover, it is difficult to determine what actual constraints VeriBetrKV's axiom implies for the underlying storage device, and whether it is sound, in contrast to our axioms that reason about the number of corrupted bits that CRC algorithms are designed to detect.

3 Verifying crash consistency using PoWER

This section discusses our novel, tool-agnostic technique PoWER for verifying crash consistency in storage systems. Unlike prior work, it can be used to concisely verify crash consistency with only basic verifier features.

3.1 PoWER

Our main contribution is Preconditions on Writes Enforcing Recoverability (PoWER), a way to verify crash consistency based on standard Hoare logic. This is challenging because while most verification tools let one specify what a method must satisfy upon completion (postconditions), one cannot generally specify what must be true *throughout* a method's execution, and crashes can happen at any point.

Overall idea. The idea behind PoWER is to give a method that performs durable updates a storage handle with a special API that ensures storage is always in a consistent, recoverable state. To do this, we add a single precondition to the API's write method, which requires all new crash states introduced by the write to be crash consistent.

```
1  pub exec fn write(&mut self, addr: u64,
2      bytes: &[u8], perm: Tracked<&Perm>)
3    requires
4      addr + bytes@.len() <= old(self)@.len(),
5      forall |s| can_result_from_partial_write(
6          s, old(self).durable_state,
7          addr as int, bytes@)
8        ==> perm@.permits(s),
9    ensures
10     self@.can_result_from_write(old(self)@,
11       addr as int, bytes@)
```

Listing 1: A simplified signature of an asynchronous `write` method used with PoWER in Verus

This is possible because one can describe all new crash states introduced by a write *before* it is invoked, even though asynchronous partial completion of those writes can occur at any time. By adding a precondition to the write method that forces the developer to reason about all such resulting crash states, we can ensure the developer cannot introduce crash-consistency bugs.

Modifying the API this way ensures correctness with no performance cost because all annotations involved in ensuring preconditions on writes are erased at compile time. The compiler generates an executable equivalent to what would exist with the standard, non-PoWER API.

The PoWER API. A standard storage API includes three main methods: `read`, which returns the most recently written bytes at a given address; `write`, which starts an asynchronous write but does not necessarily make it immediately durable; and `flush`, which ensures prior writes are durable. Their standard preconditions check properties like in-bounds addresses.

Listing 1 shows a simplified specification of a `write` method in the PoWER API in Verus. The `requires` clause specifies preconditions and the `ensures` clause specifies postconditions. `old(x)` is the contents of mutable reference `x` upon method invocation. `x@` is shorthand for `x.view()` and represents an abstraction of `x` (e.g., the current state of a storage device including outstanding writes). The new precondition introduced by PoWER is on lines 5–8; it requires that all newly introduced potential crash states are permitted. In Verus, we express the set of permitted states with an unforgeable ghost (i.e., erased by the compiler) permission token `perm`. The set of newly introduced potential crash states is defined by the storage model, which accounts for device properties like atomic write granularity and alignment; we now discuss that model.

Storage model. The storage model we use is based on the prophecy-based asynchronous disk model [62] used in Perennial [6,7]. This model eases reasoning about crashes by letting

```
pub struct PersistentMemoryRegionView { pub read_state: Seq<u8>, pub durable_state: Seq<u8> }

pub open spec fn chunk_corresponds(s1: Seq<u8>, s2: Seq<u8>, chunk: int) -> bool {
  forall |i: int| 0 <= i < s1.len() && i / chunk_size() == chunk ==> s1[i] == s2[i]
}
pub open spec fn can_result_from_partial_write(
  post: Seq<u8>, pre: Seq<u8>, addr: int, bytes: Seq<u8>) -> bool
{
  post.len() == pre.len() && forall |chunk| {
     ||| chunk_corresponds(post, pre, chunk)
     ||| chunk_corresponds(post, update_bytes(pre, addr, bytes), chunk)
}}
impl PersistentMemoryRegionView {
  pub open spec fn can_result_from_write(self, pre: Self, addr: int, bytes: Seq<u8>) -> bool {
    &&& self.read_state == update_bytes(pre.read_state, addr, bytes)
    &&& can_result_from_partial_write(self.durable_state, pre.durable_state, addr, bytes)
}}
```

Listing 2: Part of the specification used by the prophecy-based asynchronous storage model in Verus

proof code reason about possible future system states, akin to prophecy variables used in some refinement proofs [34, 42]. The Perennial authors have proven, in Rocq, that any system proven correct using the prophecy model is also correct using a natural model that cannot see the future [61]. We have also formally proven the soundness of our prophecy-based model by building it as a library atop a similar natural non-prophecy model, leveraging support for prophecy variables in Verus.

To explain the prophecy model, we first describe the natural model. Storage is divided into chunks with size equal to the atomic persistence granularity of the device. For instance, persistent memory uses a granularity of 8 bytes, while hard drives generally use 512 B or 4 KiB sectors. A read returns the last-written contents, even if the asynchronous writes that wrote them are still outstanding and may never become durable. A flush completes all outstanding writes. On a crash, each outstanding write operation is divided into chunk-granularity subwrites and some of the subwrites are durably performed. This model is tricky to reason about because the state includes both a current readable state and a set of outstanding writes.

The prophecy model, given in Verus in Listing 2, is simpler in that the state consists of only two byte sequences: the read state and the durable state (line 1). The read state reflects all writes performed so far, including outstanding updates that may be lost in a crash. The durable state reflects all subwrites performed so far that will *eventually* become durable, due to either a subsequent flush or a crash that will nondeterministically choose to render them durable.

A write operation applies the entire write to the read state and a nondeterministically chosen subset of chunk-granularity subwrites to the durable state. This is shown in Listing 2, with `can_result_from_write` (lines 13–15) specifying the possible prophesized results of a given write operation.

The flush operation's postcondition says that the read state matches the durable state, and that neither of these is changed by the flush. That is, the durable state does not *become* the read state as a result of the flush; rather, the flush confirms that the prophesized durable state matches the read state. The intuition for why this is valid is that when reasoning about instructions past a flush, one need not consider possible branching timelines in which some subwrites were lost.

We switched from the natural model to the prophecy model partway through development of CapybaraKV and found that it made proving crash consistency much simpler. One must still reason, when calling a PoWER `write`, about all the possible new crash states the write can introduce. But after the write, thanks to the prophecy model, one need only reason about the single prophesized resulting durable state.

Specifying prophecy variables in a storage model requires the use of ghost variables, which are a feature of most verification tools including Dafny and Verus. Ghost variables can contain information known at verification time even if it cannot necessarily be known during execution, which is what makes them a standard tool for prophecy variables.

Specifying crash-consistent states. PoWER, like all crash-consistency specifications [7, 9], requires that the specification writer formally specify the set of crash-consistent states. In theory, the developer may define this set however they like. We suggest the following approach, exemplified by Listing 3 which shows a simplified signature for a synchronous, crash-atomic append operation in an append-only log. Have the code define a recovery function `rec` that maps a sequence of bytes to an abstract state. Then, make the set of crash-consistent states be the union of two sets: (1) the set of states abstractly equivalent to the initial state, and (2) the set of states permitted by the postcondition. In other words, when crashing mid-operation, the code may either atomically execute that operation or do nothing.

Specifying crash-consistent transitions. Another way to specify crash-consistency properties is to specify a set of

```
pub exec fn log_append(&mut self, ps: &mut PoWERStorage, bytes: &[u8], perm: Tracked<&Perm>)
  requires forall |s| Self::rec(s) == Self::rec(old(ps)@) || Self::rec(s) == Self::rec(old(ps)@) + bytes@
    ==> perm@.permits(s),
  ensures Self::rec(ps@) == Self::rec(old(ps)@) + bytes@,
```

Listing 3: Signature for a log-append method that enforces crash consistency with PoWER

```
pub exec fn log_append(&mut self, ps: &mut PoWERStorage, bytes: &[u8], perm: Tracked<Perm>)
  requires forall |s1, s2| Self::rec(s2) == Self::rec(s1) + bytes@ ==> perm@.permits_transition(s1, s2),
  ensures Self::rec(ps@) == Self::rec(old(ps)@) + bytes@,
```

Listing 4: Signature for a log-append method that uses a single-use PoWER permission permitting a transition

legal *transitions* between crash states. That is, a permission can express not a set of legal crash states but a set of legal *pairs* of states (s_1, s_2). If the current durable state is s_1 then it permits introducing new crash state s_2.

A useful case of this is a permission to transition from any s_1 to s_2 if they both recover to the same abstract state. Another case is that for a mutating operation, one can grant permission to transition from s_1 to s_2 if s_2 recovers to a state that matches s_1 except that the operation has been performed. Listing 4 shows an example of this use.

Blanket vs. single-use permissions. Some scenarios call for a *blanket* permission, i.e., a permission that can be used repeatedly. A key case of this is permission to transition between recovery-equivalent states. Such permission can be granted to the system when it starts, so that it can make such transitions throughout its lifetime. This simplifies the specification by reducing the number of permissions that must be granted. Other scenarios call for *single-use* permissions. A key case of this is permission to perform a transition corresponding to a state-mutating operation that should only happen once.

Single-use permissions are straightforward to implement, by having a `write` operation consume or otherwise invalidate any single-use permission passed to it. In such an implementation, a blanket permission is a factory that can generate an arbitrary number of single-use permissions.

Statically partitioning into multiple PoWER regions. For some systems, it is convenient to statically partition storage into multiple parts, each with its own PoWER handle. We call each such part a *region*. Such partitioning can be done by using multiple physical or virtual devices, or by treating each file in a file system as a separate region. This modularizes reasoning about storage state since each region becomes a separate object with its own crash constraints and permissions.

3.2 Correspondence to other approaches

To demonstrate the soundness of the PoWER approach for specifying crash consistency, we use mechanically-checked proofs to show correspondence to two other approaches: CHL and crash invariants. These are among the current state of the art for verified storage systems, but require additional verifier features beyond Hoare logic and are thus not tool-agnostic.

Correspondence to Crash Hoare logic. To validate the soundness of our PoWER specification approach, we produce a mechanically-checked Rocq proof of its correspondence to CHL. We prove that any code satisfying a PoWER specification satisfies a corresponding CHL specification, encoded as a *crash weakest precondition* (WPC) in Perennial, whose crash condition states that, if the system crashes, the storage will satisfy the recoverability predicate. The details of this theorem statement, along with the proof, are available as part of this paper's artifact, described in §A.

Since we have not implemented PoWER in Rocq, this proof is metalogical and depends on a trusted translation of PoWER semantics into Rocq. This translation is fairly natural, so we feel confident in the correctness of this proof.

Correspondence to crash invariants. Crash consistency can also be specified using crash invariants, as in Perennial [6]. We prove that satisfying a PoWER specification implies the satisfaction of a crash invariant, in two steps. First, we implement the PoWER interface in Verus as a verified library on top of a lower-level interface that reasons about crash consistency using crash invariants. Second, we prove that, if an application adheres to the PoWER API, it implies that the PoWER library maintains a corresponding crash invariant on behalf of the application. We can do this because crash invariants can be built on top of atomic invariants, an advanced feature of some verification tools including Verus but not Dafny, Prusti [65], or Creusot [13]. Since both the PoWER interface and the invariant-based specification are in Verus, this proof is entirely machine-checked (unlike the CHL proof mentioned above). The details of this correspondence are available as part of this paper's artifact, described in §A.

3.3 Strategies for satisfying preconditions

We next discuss how a developer can prove their code matches a PoWER specification. The challenge is proving, immediately before each `write`, that all new crash states that can result from partial application of the write are permitted. In this subsection, we describe four design patterns that simplify this reasoning, and libraries we provide that make it easy to apply them. We classify durable updates into four categories—tentative, committing, recovery, and in-place—and provide

strategies to prove the crash consistency of each category.

Tentative writes. We call a write *tentative* if it has no effect on the abstract system state until some subsequent write happens. Tentative writes generally modify data at addresses that are unreachable during recovery (e.g., inodes unreachable from a file system's root), and do not change system state regardless of which subwrites become durable. Their contents only become relevant after a subsequent, non-tentative write (e.g., storing a reachable pointer to the address).

The developer need not prove anything about the specific bytes in a tentative write to prove crash consistency. They must only prove that the addresses modified by the write are unreachable during recovery. We provide lemmas in our Verus and Dafny libraries (see Listing 5 for the Dafny version) that the developer can call to satisfy the precondition of the PoWER API.

Committing writes. We call a write *committing* if it changes the abstract state of the system using a single crash-atomic write. For example, such a write might update a pointer that causes a tree of objects on storage to become reachable by the recovery function. A committing write is typically done after a flush to ensure that a crash does not cause the state to be invalid due to lost tentative writes.

For committing writes, the developer only needs to reason about two possible crash states: the states that result from the committing write being dropped or applied. We provide, in our library, a lemma that ensures that for a committing write there are only these two possible crash states.

Recovery writes. *Recovery writes* are writes done as part of a recovery procedure (e.g., replaying a journal). Such writes are not tentative, as they can change the abstract view, and they are not committing, as they may not match the atomic write granularity. To prove that such writes are crash consistent, a developer must prove that they only modify bytes that will be written to by a completed recovery procedure, and that recovery is idempotent. If the system crashes while recovering from an earlier crash, all modifications made during the first recovery will be overwritten by the second recovery, ensuring that any torn writes are fixed when recovery completes.

We have written a generic redo journal component in Verus. Internally, this journal must reason about intermediate crash states, but the user of the journal component (e.g., CAPYBARAKV) does not have to. The user just logs updates and eventually atomically commits (or aborts) the updates gathered in the log. The journal takes care of installing journaled operations on commit or when replaying its log after a crash.

In-place writes. *In-place writes* non-atomically modify user-visible state and can change the abstract state of the system. They thus leave the system in a nondeterministic abstract state. This may be reasonable for systems that provide weak crash-consistency guarantees, e.g., a file system that lets a read see a write that is then lost by a crash. Such a specification would permit a large set of possible crash states, so in-place writes could more easily be proven to produce states in that set.

We have not yet had experience verifying storage systems with weakly crash-consistent semantics, so we currently have no support in our libraries for reasoning about in-place writes. A developer of such a system would have to directly prove the PoWER preconditions.

3.4 Extending PoWER to concurrent contexts

So far, we have described how PoWER can verify non-concurrent storage systems, building on Hoare logic. Some verification tools have additional features for concurrency reasoning, and while PoWER does not require such features, it can leverage them to reason about some forms of concurrency.

PoWER can easily support systems that issue concurrent reads to storage, such as an implementation that uses a reader-writer lock to protect access. PoWER supports this pattern because it only adds a precondition for write operations and does not change how reads are done.

PoWER can also support systems that concurrently issue reads and writes to multiple statically allocated storage regions, as long as there is never read-write or write-write concurrency on a single such region. An example of this is a KV store sharded over multiple storage regions. To enable proving a linearizable specification for such a system, we introduce mechanisms to formally reason about the linearization order of operations across multiple storage regions. To do this, we extend PoWER to support Perennial-style reasoning about crash safety using atomic invariants, which are also supported in Verus. Specifically, we make the following two extensions to PoWER, which together we call *atomic PoWER*.

First, we introduce ghost state representing the durable state of persistent storage, based on Perennial's notion of durable resources. These durable resources can be owned by an atomic invariant, which enforces that some application-level recovery predicate always holds on the durable state (and in particular, this invariant can refer to the durable state of multiple storage regions), and therefore this invariant will be true when the system crashes and initiates recovery.

Second, we introduce the notion of a *completion* object returned when a write consumes a single-use permission. Logically, this captures the fact that a write matching the permission's `permits_transition()` was issued, and lets an application prove that its operation was applied to durable storage. For example, when an application invokes a KV store with a permission to create a new key, the KV store can return a completion proving that it wrote the key to storage, even if some concurrent thread has since deleted that key.

In atomic PoWER, permissions are no longer a trusted object but rather a verified wrapper around a Perennial-style proof (in Verus, a proof-mode callback) that opens the atomic invariant holding the durable resource and updates it when the write is applied. With this change, PoWER's API takes on the form of a *logically atomic crash specification* from

```
ghost predicate AddressesUnused<T>(s: seq<byte>, addrs: set<int>, rec: seq<byte> -> T) {
  forall s2: seq<byte> :: |s2| == |s| &&
    (forall i: int :: 0 <= i < |s| && i !in addrs ==> s[i] == s2[i]) ==> rec(s2) == rec(s)
}
lemma Lemma_TentativeWritePermitted<T>(ps: PoWERStorage, addrs: set<int>, rec: seq<byte> -> T,
                                       bytes: seq<byte>, start: int)
  requires forall s :: rec(s) == rec(ps.View().durableState) ==> s in ps.StatesPermitted()
  requires AddressesUnused(ps.View().durableState, addrs, rec)
  requires forall addr: int :: start <= addr < start + |bytes| ==> addr in addrs
  ensures  forall s :: CanResultFromPartialWrite(s, ps.View().durableState, start, bytes) ==>
           s in ps.StatesPermitted()
```

Listing 5: Dafny library lemma for proving that a tentative write satisfies the PoWER write API

Perennial [7, §6.2]. As a result, the application developer's job becomes to first state an atomic invariant about the durable ghost state, then to implement (perhaps using the library discussed in §3.2) permissions for any write operations they want to issue via a PoWER API. Intermediate layers between the top-level application atomic invariant and the low-level storage device are unaffected by these changes. For instance, we can reuse nearly all of CapybaraKV's single-threaded code for its concurrent variants because that reusable code satisfies a PoWER API and thus does not care how permissions it receives are implemented (see §5.1.2).

Limitations. The key limitation of PoWER's design, in terms of concurrency, is that it cannot reason about writes executing concurrently with other reads or writes to the same storage region. The constraint comes from the fact that the PoWER API requires the caller to logically know the current state of the region (`old(self)@` in Listing 1) at the time a `read` or `write` operation is issued. However, due to concurrent writes, the caller may not fully know that state.

4 Provably detecting corruption

Stored data may become corrupted over time due to media errors, so checking the integrity of data using cyclic redundancy checks (CRCs) is standard in many storage systems [4, 16, 66, 67]. A verified storage system should require that data read from the storage device is checked for corruption before it is used or returned to the user. In this section, we introduce a new model of media corruption, and a new corruption-resistant atomic primitive for persistent memory.

4.1 Modeling media corruption

To model possible data corruption, the postcondition of our `read` method specifies that the returned bytes may not match the last-written bytes, but rather are related to them via a `maybe_corrupted` predicate shown in Listing 6. The developer must perform a CRC check to prove that the returned bytes are uncorrupted before using them.

Our model of device corruption is as follows. The device has a corruption bitmask with one bit per storage bit. Where the bitmask is 0, reads return correct data. Where it is 1, reads return arbitrary bits, not necessarily the same on each read. The population count of the bitmask, i.e., the number of 1s, is bounded by a constant c that is opaque to the code to be verified. However, that code has access to a trusted fast CRC library [12] and an axiom stating that any two byte sequences with Hamming distance in $[1, c]$ have different CRCs. (For the ECMA variant of CRC-64 that we use, $c = 1$ for arbitrary-length data, i.e., it guarantees to catch any single-bit errors, but c can be higher for shorter lengths, such as $c = 3$ for data shorter than $\sim$1 GiB [39]. Our implementation currently does not take advantage of the higher c values for shorter lengths.) This means that, assuming c or fewer device bits are corrupted, the result of a CRC check on a given buffer definitively proves whether or not the buffer has been corrupted.

This model differs from VeriBetrKV's "corruption cannot produce a block with a valid checksum" [21]. Our model is more fundamental, describing the behavior of the media at a lower level. It is also more flexible, allowing the contents protected by a checksum to be noncontiguous and to not be in the same block as the checksum. This flexibility is required when building PM systems like CapybaraKV and CapybaraNS, as we discuss next.

4.2 Checking for PM corruption

Persistent memory presents new challenges for maintaining CRCs for corruption detection. Traditional storage systems often store a CRC of each block's contents within the block itself [58]. However, PM's finer write granularity (8 aligned bytes [64]) makes this technique crash-unsafe, as the hardware does not guarantee that the CRC will be written atomically with any non-trivial amount of data.

Prior work [44] on testing NOVA-Fortis [67], a corruption-resistant PM file system, shows that CRC management on PM is prone to crash consistency bugs. Furthermore, when trying to use and verify NOVA-Fortis's *Tick-Tock* algorithm for updating CRCs, we find it is not correct in either our model of corruption or VeriBetrKV's [21]. Tick-Tock maintains two

```
pub open spec fn maybe_corrupted(self, bytes: Seq<u8>, true_bytes: Seq<u8>, addrs: Seq<int>) -> bool {
  &&& bytes.len() == true_bytes.len() == addrs.len()
  &&& forall |i: int| 0 <= i < bytes.len()
    ==> exists |mask: u8| bytes[i] == (true_bytes[i] ^ (mask & self.corruption[addrs[i]]))
}
pub exec fn read(&self, addr: u64, num_bytes: u64) -> (bytes: Vec<u8>)
  requires self.inv(), addr + num_bytes <= self@.len(),
  ensures ({
    let true_bytes = self@.read_state.subrange(addr as int, addr + num_bytes);
    let addrs = Seq::<int>::new(true_bytes.len(), |i: int| i+addr);
    self.constants().maybe_corrupted(bytes@, true_bytes, addrs) })
```

Listing 6: Read method specification describing possible corruption of returned bytes

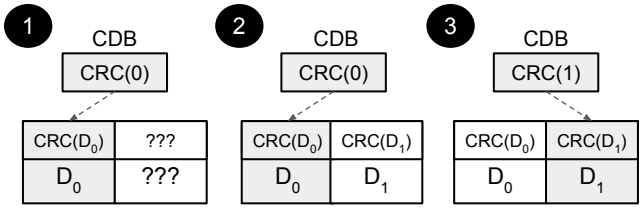


Figure 1: One way to use a CDB to atomically update a data structure and its CRC

copies (a primary and a replica) of each persistent data structure, each with its own CRC. To update the data structure, it updates the primary, then flushes, then updates the replica. On recovery, it uses whichever copy has a matching CRC, preferring the primary. However, CRC algorithms are designed to defend against media corruption in the form of a positive and bounded number of bit flips, not to distinguish between different user-level values. CRC collisions between such values, which can differ arbitrarily, are plausible (and indeed likely in scenarios where adversaries can manipulate values).

For example, suppose we begin updating a data structure with value D_0 to D_1. Tick-Tock writes D_1 and $\mathsf{CRC}(D_1)$ to the primary; suppose a crash occurs after D_1 becomes durable but before $\mathsf{CRC}(D_1)$ becomes durable, so the primary CRC remains $\mathsf{CRC}(D_0)$. Also suppose that the stored version of D_1 is corrupted into D_1' and that, by chance, $\mathsf{CRC}(D_0) = \mathsf{CRC}(D_1')$. Tick-Tock's primary CRC check will pass and it will not detect the corruption. This is plausible in both our and VeriBetrKV's models, since D_1' and D_0 may have an arbitrary number of bit differences.

To address this challenge, we propose the following new primitive that enables atomic updates on PM. The *corruption-detecting Boolean* (CDB) is an 8-byte integer that can only take on two specific values, one representing `false` and one representing `true`. These two values should be chosen carefully such that neither is likely to be corrupted into the other; we use $\mathsf{CRC}(0)$ and $\mathsf{CRC}(1)$. Since a CDB is 8 bytes, it can be written to PM atomically with respect to crashes. Since its valid values are statically known, it can be checked for corruption without needing to maintain a separate CRC.

Here is one way to use a CDB to implement an atomically mutable data structure D. Reserve space for an 8-byte CDB and two versions of D plus their CRCs. The CDB indicates which version is considered valid at recovery time. To update, tentatively write a new version and its CRC to the invalid location, then flush, then use a committing write to flip the CDB, then flush. Figure 1 illustrates this for the case where we start in step ❶ with D_0 as the valid version. In step ❷ we write the new version to D_1 and its CRC to $\mathsf{CRC}(D_1)$, then flush. In step ❸ we update the CDB to $\mathsf{CRC}(1)$ then flush. For the next update (not shown in the figure), we will go the other way: store the new version and CRC in D_0 and $\mathsf{CRC}(D_0)$, then flush, then update the CDB to $\mathsf{CRC}(0)$.

We find the CDB to be a useful primitive, and use it in several places in CapybaraKV and CapybaraNS to facilitate atomic updates. For instance, we use it in CapybaraNS to atomically update the counter and last hash, using the steps in Figure 1. To our knowledge, ours is the first proven-correct algorithm for atomic updates on PM with corruption detection.

5 Verified systems

To demonstrate our support for multiple verification tools, we implement CapybaraKV, the first verified PM key-value (KV) store, in Verus and CapybaraNS, the first verified persistent notary service, in Dafny.

5.1 CapybaraKV

CapybaraKV is an embedded PM key-value store with verified functional correctness, crash consistency, and corruption detection. It maps keys to values, where each value consists of an item and a list of elements. It is parameterized by key, item, and list-element types, with each of these types having a fixed size. It supports standard create, read, update, and delete operations on key-item pairs, as well as operations that update the list associated with a key. These list operations can append an element, trim a given number of elements from the front, or update the element at a given index. CapybaraKV supports crash-atomic transactions in which operations are visible immediately but not durable until committed.

Trait	Description	Generated code
`PmSafe`	Ensures safety of copying to storage	No methods; trivial `PmSafe` trait bounds on field types
`PmSized`	Ensures size known at verification time	Spec and exec `size_of` and `align_of` methods, and static assertions that their output matches compiler-generated type layout
`Clone`	Implements explicit copy method	Specification that copy equals original
`PartialEq`	Implements equality operator	Specification that operator is consistent with Verus equality

Table 1: Traits implemented by the `#[derive(PmCopy)]` macro in the `pmcopy` crate

Specification. Its abstract state is two maps from keys to values: the on-abort map and the on-commit map. The on-abort map represents what should result if an abort is requested or the system crashes. The on-commit map represents what should result if a commit completes. The abstract state's operations include abort, commit, create, read, update, delete, and various list-updating operations. The abort operation sets the on-commit state to the on-abort state, representing both the expected behavior if a transaction aborts or if the system crashes. The commit operation sets the on-abort state to the on-commit state, and every other operation affects only the on-commit state. In the event of a crash, the commit operation may either abort or commit the current transaction; all other pending operations must abort.

Implementation. CapybaraKV has four main durable components: a main table, an item table, a list-element table, and a journal. Each row in the main table contains a key, the address of the corresponding row in the item table, the address of the row in the list-element table containing the first list element, a CRC of the key, a CRC of the address fields, and a CDB (§4.2) indicating if the row is valid. Each row in the item table contains an item and its CRC. Each row in the list-element table contains a list element, the address of the next item in the list if any, and a CRC of each of these fields. An item row is considered valid if it is pointed to by a valid main table row. A list-element row is considered valid if it is pointed to by a valid main table row or list-element row.

CapybaraKV uses a physical redo journal. When a transaction is committed, we tentatively append a single CRC for all pending entries, then commit by updating a CDB. Log entries are replayed using recovery writes. Once all entries have been installed, we clear the journal. The journal is designed to be of general use beyond CapybaraKV.

New records are created in CapybaraKV by tentatively writing new main and item table entries to free rows, then journaling an update to the main table entry's CDB. Deletions only require journaling the invalidation of the target's CDB. CapybaraKV uses copy-on-write to update an existing item or list element, i.e., it allocates a new row in the appropriate table and journals an update to the row that points to it.

CapybaraKV also has a volatile index, implemented as a Rust `HashMap`, that maps all keys to their main table addresses. Each table also maintains a volatile free list used for allocation. These volatile structures are rebuilt at startup after log replay.

5.1.1 Safe reads and writes

In PM systems like CapybaraKV, PM's low access latency makes minimizing overhead essential. In particular, I/O latency can easily be eclipsed by software overheads, so many systems `memcpy` data structures between DRAM and PM with no serialization. However, such low-level operations risk crash safety and can lead to undefined behavior. For example, structures that contain references (e.g., file handles or virtual addresses) cannot safely be stored on PM, as a reference may be invalid after a crash. And, when reading stored data, we must ensure that data is placed in a properly laid-out buffer and checked for corruption before casting to a more useful data structure to avoid undefined behavior.

Unfortunately, these properties are difficult to verify because compiler-generated type and layout information is not available to verifiers. We tackle this issue by using the powerful Rust compiler to check properties that Verus cannot. We are inspired by Corundum [26], a Rust crate that uses various Rust language features to enforce safety properties in PM storage systems, and SquirrelFS [44], a PM file system with crash-consistency properties checked by the Rust compiler.

We have developed `pmcopy`, a trusted Rust crate that provides a macro to help developers check these crucial safety properties. It generates Verus ghost code to facilitate proofs that rely on type layout information and provides executable functions specified by this ghost code. It also adds compile-time assertions that are checked by the Rust compiler, not Verus, to check that axioms it synthesizes match compiler-generated information. CapybaraKV uses `pmcopy` to enforce safety properties about all durable data structures. We expect `pmcopy` will also be valuable in the development of other PM storage systems in Verus.

To use `pmcopy`, a developer need only include two annotations on the definitions of durable data structures. The first is the directive to use the C representation (`#[repr(C)]`), as the default Rust representation is intentionally under-specified and is not safe for operations that rely on a known type layout [56]. The second is `#[derive(PmCopy)]`, which causes `pmcopy` to generate an implementation of the `PmCopy` trait and several supertraits. Routines that copy to and from PM require that their parameters implement this trait. Table 1 de-

scribes four supertraits of `PmCopy` implemented by `pmcopy`. We assume that all Rust numeric types, `bool`, and `char` are safe to store on PM and mark them `PmSafe`. `PmCopy` can only be derived on structs and enums made up of `PmSafe` types.

Partway through development of CapybaraKV, an update to the Rust compiler changed the layout of `u128` and `i128`, which caused an inconsistency between compiler- and `pmcopy`-generated layouts. Thanks to the static assertions `pmcopy` generated, this discrepancy was immediately flagged.

5.1.2 Concurrent variants

In addition to the single-threaded CapybaraKV, we build two concurrent variants, one that uses a reader-writer lock to enable concurrent reads and one that uses sharding to also enable concurrent writes, representing the two forms of concurrency described in §3.4. The API shared by the two variants is a linearizable specification that is formalized using logical atomicity [32, 60] and encoded using proof callbacks in Verus. This shared API is non-transactional, consisting of single atomic operations that create, read, update, etc.

All three implementations use a single component, `UntrustedKvStoreImpl`, to implement the main KV-store functionality. This includes nearly all of the code, including setup, recovery, transactions, and individual operations. That component exports a transactional API that uses atomic PoWER. That is, it only stores and recovers state via a `PoWERStorage` object supplied by its wrapper, so it cannot perform writes unless granted permission for all possible resulting crash states. That wrapper, either the single-threaded CapybaraKV or one of its two concurrent variants, creates permissions and passes them to the `UntrustedKvStoreImpl` as needed. The sharded variant uses multiple instances of `UntrustedKvStoreImpl`, one per shard, each using a separate PM file as a storage region.

At startup time, each wrapper provides a blanket permission allowing any number of writes that do not change the abstract post-recovery state. The `UntrustedKvStoreImpl` saves this for repeated use throughout its lifetime, for recovery and for all operations except `commit`. For `commit` operations, the wrapper supplies a one-shot permission permitting transitioning from the on-abort state to the on-commit state.

The reuse of `UntrustedKvStoreImpl` demonstrates a development advantage of using PoWER for systems with coarse-grained concurrency like the two we describe here. Nearly all the code, including all the complex mechanisms for recovery and transactions, can be written within components implementing a PoWER specification. As a result, most of the time, developers only need to think about how to satisfy that simple specification. They only have to devise atomic invariants and reason about crash-consistent concurrency outside of the component, in the relatively small wrappers.

5.1.3 Discussion

CapybaraKV is designed for a particular use case, storing small keys, items, and list elements on a small amount (tens of GiB) of dedicated PM, in a production cloud storage service. In targeting this use case, we make several simplifying design decisions that streamline both implementation and verification but impose limitations on functionality.

CapybaraKV requires users to statically allocate storage space and specify at initialization the maximum number and size of keys, items, and list elements. It does not currently support dynamic resizing and will waste space if the number of records is smaller than initially specified. CapybaraKV uses a volatile index that keeps all keys in memory, which grows its memory footprint (especially if large keys are used) and must be rebuilt each time the system is started.

5.2 CapybaraNS

CapybaraNS is a notary service similar to the verified notary in Ironclad Apps [23]. It securely assigns logical timestamps to hashes so they can be conclusively ordered, and stores its state on persistent storage. We build and verify it in Dafny, with a trusted C# wrapper that provides external methods for CRCs, cryptography, and serialization.

Its abstract state consists of a current logical timestamp (a 64-bit unsigned integer) and a last hash. Its interface has two main operations: (1) `Advance` increments the timestamp and updates the last hash. (2) `Sign` uses the service's private key to sign a binding between the last hash and the timestamp. CapybaraNS uses the CDB algorithm from §4.2 to atomically update its storage state during an `Advance` operation.

5.3 Persistent memory model

We have written a model of PM in Verus and Dafny and use it to reason about potential crash states in CapybaraKV and CapybaraNS. This model uses the prophecy approach described in §3.1 and represents a PM resource using two sequences of bytes, a read state and a durable state. Our model is consistent with a storage API implementation that provides `write` and `flush` methods such that 8-byte chunks of a buffer written with `write` may be reordered until the next `flush` call. In our implementation, `write` calls a PMDK [30] function that uses non-temporal stores or `mov` followed by `clflushopt` or `clwb`, and `flush` performs an `sfence`.

There are two situations in which this model may overapproximate crash states. First, our model considers reorderings that are impossible if the implementation uses the strongly-ordered `clflush` instruction to flush cache lines. `clflush` is less performant than the weakly-ordered instructions described above, so typically PMDK will not use it. Even if it does, the instructions PMDK uses for each write are not known at verification time, so our model considers these reorderings to ensure it does not underapproximate crash states.

	Trusted	Spec+Proof	Impl
CapybaraKV			
PoWER framework	1698	1079	138
pmcopy crate	964	0	0
Journal	0	2234	611
Concurrency layer	1142	473	620
Sharding layer	395	265	332
KV store	1045	10204	3830
Total	5244	14255	5531
CapybaraNS			
PoWER framework	266	118	4
Notary server	148	555	274
Total	414	673	278

Table 2: Lines of code in each verified system

Second, our model considers states including those that are only reachable by reordering 8-byte chunks on the same cache line. Current Intel systems guarantee that such updates will not be reordered by hardware, making some of these crash states impossible in practice [55]. We choose the current model because it is simpler and more portable than a model that is tied to specific instructions or low-level hardware guarantees. It also accounts for the possibility that the compiler may reorder independent `mov` instructions.

We note that PoWER is not tied to any particular storage model, so these overapproximations are not fundamental to the technique. A developer could use a different model for a system that always uses `clflush` and/or that never causes reorderings on the same cache line.

6 Evaluation

This section addresses the following questions: How much effort does it take to build and verify a new system with PoWER (§6.1)? How does CapybaraKV compare to similar, but unverified, PM key-value stores (§6.2)?

6.1 Verification effort

Table 2 gives the number of lines of code in major components of CapybaraKV and CapybaraNS, organized into trusted (i.e., unverified), specification/proof, and executable code. We count the unverified `pmcopy` crate towards CapybaraKV's trusted code, but we do not count the lines of code it generates. The CapybaraKV PoWER trusted line count also includes a mock PM region using a byte vector and PM backends for Windows and Linux. Both systems have a low proof-to-code ratio (2.6 for CapybaraKV and 2.4 for CapybaraNS).

Designing, implementing, and verifying CapybaraKV took approximately 1.5 years of work by a team consisting of both verification experts and newcomers. We built CapybaraNS when CapybaraKV was mostly complete, so its development benefitted from lessons learned when building CapybaraKV. It took less than one person-hour to port the PM specification to Dafny, about one hour to port the library supporting reasoning about tentative and committing writes, and about nine hours to implement and verify CapybaraNS in Dafny after writing its specification and C# wrapper.

Verification time. On one of our development machines (Linux v6.9.3, Intel Core i7-11850H CPU, 8 physical cores, 32 GB memory), it takes 54 seconds to verify CapybaraKV with 1 thread and 23 seconds with 8 threads. It takes 12 seconds to verify CapybaraNS with 1 thread; Dafny does not support multi-threaded verification.

6.2 CapybaraKV performance

We evaluate CapybaraKV against three unverified PM key-value stores: pmem-Redis [28], pmem-RocksDB [29], and Viper [3]. CapybaraKV is the first verified PM KV store, so we cannot compare to prior verified systems. VeriBetrKV [21], the most similar verified system, is designed for block devices.

Experimental setup. We run experiments on a two-socket machine with 32 physical cores, 128 GB memory, and one 128 GB Intel Optane DC PMM. The evaluation machine runs Debian Trixie and Linux 6.12.10. We run the pmem-Redis server and client on the same machine, enable its pointer-based append-only file, and configure it to store the AOF and all values on PM. We configure pmem-RocksDB to memory-map files for reading and writing and to use non-temporal stores when appending to its log. We set Viper to preallocate enough space for each experiment that dynamic resizing does not impact its performance. We use the sharded variant of CapybaraKV for all experiments, with one shard per thread.

Microbenchmarks. Figure 2(a) gives the average latency of various operations on items in the four evaluated KV stores with both sequential and random access patterns. Figure 2(b) gives the average latency of operations on lists in pmem-Redis and CapybaraKV; the other systems do not support lists.

Each experiment uses 25M records with 64-byte keys, 1 KiB values, and 8-byte list elements. The list "append" experiment appends a total of 16 list elements to all records, accessing the records in a random order. The "read list" operation returns the entire contents of the list as a vector, and "trim" removes all elements from the list. All operations except for list append are run once on each record.

Pmem-Redis has the highest latency on all operations due to communication overhead between its client and server. CapybaraKV achieves similar or better latency to pmem-RocksDB on all measured operations primarily due to its tiny (< 64 B) journal entries and its fast hash-map index. Pmem-RocksDB spends more time appending larger entries to its write-ahead log and managing its smaller but more

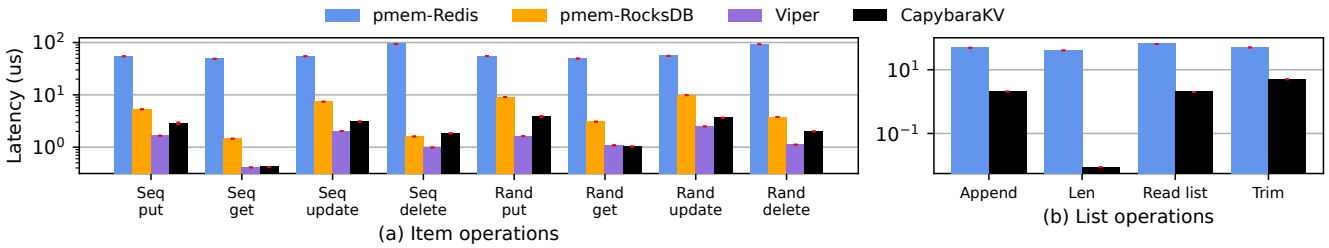


Figure 2: Average operation latency in microseconds. Note the log scale. Error bars show 95% confidence intervals.

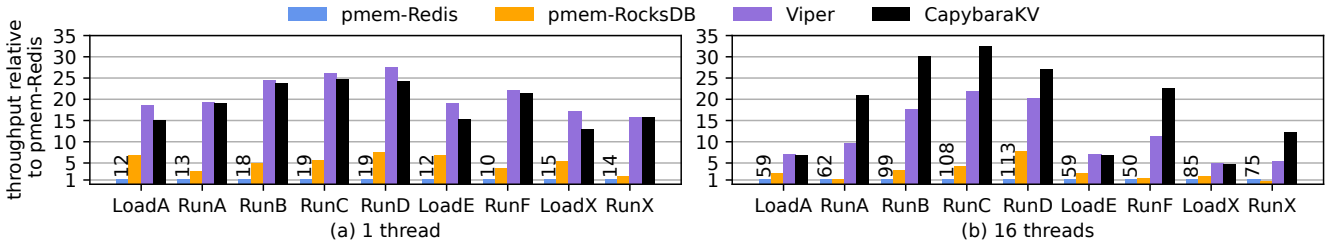


Figure 3: YCSB throughput relative to pmem-Redis. Numbers above pmem-Redis bars show absolute throughput in kops/s.

complicated in-memory MemTable and cache structures. Key lookups in pmem-RocksDB involve searching the MemTable and potentially multiple durable files.

CapybaraKV's random get latency is approximately 2× its sequential get latency in this microbenchmark because sequential loads are faster than random loads on Optane PM [31]. The sequential get workload runs on records that were inserted sequentially, whereas random get accesses keys that were inserted in a different random order.

Battery-backed DRAM. We also evaluate CapybaraKV in a testing environment in Azure Storage running Windows with 20 GiB battery-backed DRAM. We run the microbenchmarks shown in Figure 2(a) on CapybaraKV in this environment and find that operations are up to 2× faster on battery-backed DRAM and follow similar performance patterns. We are unable to evaluate the other systems in this environment as they do not support Windows.

Macrobenchmark: YCSB. We also measure each system's performance on several workloads from the widely-used YCSB benchmark suite [10]. CapybaraKV and Viper do not currently support range queries, so we skip the YCSB workload that includes them (RunE). Several workloads (RunA and RunB) involve partial value updates, which these systems also do not support, so we modify the workloads to always update full values. We also introduce workload X, which is based on a trace of traffic to a production service similar to CapybaraKV at Azure Storage and consists of 75% updates, 5% read-modify-write operations, and 20% reads with a uniform access distribution. All YCSB workloads use 15M keys and are executed 5 times on each system. The CapybaraKV instances in these experiments use 24-byte keys and 1140-byte values, both structured as byte arrays.

Single-threaded performance. Figure 3(a) gives the average throughput of each system using one thread relative to pmem-Redis. CapybaraKV significantly outperforms pmem-Redis and pmem-RocksDB and achieves similar performance to Viper on these workloads. Pmem-Redis is unable to achieve high throughput on these operations due to its high per-operation latency. Multiple factors contribute to pmem-RocksDB's lower performance: background threads performing LSM tree compaction slow down client threads; lookups and updates in its in-memory MemTable index take longer than in Viper and CapybaraKV's hash maps; pmem-RocksDB uses per-block CRCs that take longer to compute than CapybaraKV's smaller per-structure CRCs; and, unlike Viper and CapybaraKV, it logs all keys and values.

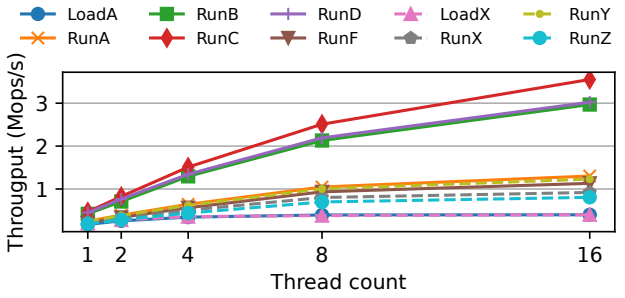


Figure 4: Sharded CapybaraKV YCSB throughput with different thread counts

Viper and CapybaraKV achieve similar performance on these workloads because they have similar architectures. Both use in-memory hash map indexes and use simple durable data structures. Viper also includes optimizations for running concurrent workloads on interleaved PM, but our experiments use a single non-interleaved NVDIMM.

Sharded performance. Figure 3(b) gives average throughput using 16 threads (the number of physical cores in each

	Startup time		Utilization (GiB)	
	Empty	Full	Memory	Storage
pmem-Redis	142 ms	-	12.3	22
pmem-RocksDB	9 ms	7 ms	2.0	17
Viper	9 s	75 s	1.1	23
CapybaraKV	7 s	53 s	2.8	18

Table 3: Startup times and memory and storage utilization on YCSB LoadA on 128 GiB Optane PM

NUMA node on our test machine) and 16 CapybaraKV shards. Each record is placed in a shard based on a hash of its key, which results in a roughly uniform distribution. We do not shard the other three systems in this experiment, as they already support multiple concurrent clients. Figure 3(b) shows that with our simple sharding protocol, CapybaraKV outperforms the other systems on concurrent workloads including, suprisingly, Viper. CapybaraKV uses coarser-grained concurrency control with lower overhead than Viper. We also find that Viper's performance in these workloads is impacted by the use of semaphores in its in-memory hash map implementation, CCEH [49]. CCEH protects hash buckets using per-bucket semaphores, and we observe via traces collected using magic-trace [27] that using these semaphores becomes slower with more workload threads (nearly doubling when going from 1 to 16 threads). CapybaraKV uses Rust's standard library `RwLock`, which does not have this issue.

Figure 4 gives average throughput of CapybaraKV on each YCSB workload with different thread counts. We omit LoadE for readability and because it has the same performance as LoadA. We introduce two new workloads, Y (50% reads, 50% updates) and Z (5% reads, 95% updates), which both use a hotspot access distribution where 90% of the operations access 10% of the records. Read-heavy workloads (Runs B, C, and D) scale well because the per-shard locks allow concurrent reads. The more write-heavy workloads experience more lock contention and do not scale as well. Since our sharding policy evenly distributes keys across shards, Y and Z perform similarly to the non-hotspot workloads.

Startup times. Table 3 compares how long it takes for each key-value store to start up on both an empty instance and a completely full instance on a 128 GiB PM device. To measure full startup times, we insert records with 64-byte keys and 1 KiB values into each system until it returns an out-of-space error, then repeatedly start each system on those records. Attempting to start the pmem-Redis server on a full instance fails with a memory allocation error. CapybaraKV and Viper take much longer to start on both empty and full systems than pmem-Redis or pmem-RocksDB because they need to initialize or reconstruct their in-memory data structures. Viper takes longer than CapybaraKV because it memory-maps more files and hashes each key before inserting it into the index.

Memory and storage utilization. Table 3 also reports DRAM and PM utilization in YCSB's LoadA workload. We use the same configurations as before for all systems but Viper, which starts with a 1 GiB pool and grows as keys are added. CapybaraKV uses more memory than pmem-RocksDB because it maintains all keys in DRAM. Viper only keeps hashes of keys in DRAM, which reduces its memory footprint. Pmem-Redis also keeps all keys in DRAM, but has high per-key overhead (almost 700 bytes as reported by the server).

18 GiB is the minimum space required by CapybaraKV to store these records; note that unlike the other systems, this CapybaraKV instance cannot grow any further. In an optimally-provisioned instance, CapybaraKV has low storage space overheads due to its simple durable layout and does not use much more storage than pmem-RocksDB. Viper uses more storage space because it checkpoints some system metadata for use in crash recovery. To achieve durability guarantees similar to the other systems, we configure pmem-Redis to store mappings from keys to durable values as well as the values themselves, which adds additional storage overhead.

7 Conclusion

This paper presents new techniques for proving crash consistency and corruption detection in storage systems. We introduce PoWER, a way to prove that a system is crash consistent using only basic verifier features like Hoare logic, and a new model of storage corruption that forces developers to properly perform corruption detection while giving them flexibility in how to do so. We develop a new primitive, the corruption-detecting Boolean, and show how to use it to implement a novel algorithm for atomic updates to PM data structures. To demonstrate that our approach is useful and tool-agnostic, we build two verified PM storage systems, CapybaraKV in Verus and CapybaraNS in Dafny. We evaluate CapybaraKV and find that its performance is competitive with unverified PM KV stores. CapybaraKV and CapybaraNS are available at https://github.com/microsoft/verified-storage.

Acknowledgments

We would like to express our great appreciation to our anonymous shepherd and reviewers for their helpful feedback on the paper. Their disappointment with a lack of concurrency support moved us to devise, implement, and verify concurrency mechanisms, which we feel have substantially improved the work. We are also grateful to Tej Chajed, Travis Hance, Jon Howell, Jialin Li, Bryan Parno, Upamanyu Sharma, and Joseph Tassarotti, with whom we had useful conversations about the work and its relation to Perennial and VeriBetrKV. Hayley LeBlanc and Vijay Chidambaram were partially funded by donations from Toyota.

A Artifact Appendix

Abstract

The artifact for this paper includes the source code for CapybaraKV and CapybaraNS and proofs of correspondence between PoWER and other crash-consistency verification techniques. It also includes instructions and code to run the experiments described in §6 and generate the corresponding tables and graphs.

Scope

The artifact can be used to validate all of the main claims of the paper. Specifically, it validates the following claims:

1. CapybaraKV and CapybaraNS are correct, crash consistent, and corruption resistant, and can be verified using Verus and Dafny, respectively;
2. CapybaraKV and CapybaraNS have low proof-to-code ratios and can be verified quickly;
3. CapybaraKV achieves competitive performance compared to state-of-the-art unverified PM key-value stores.

Contents

The artifact has three main parts:

1. The source code for CapybaraNS and instructions on how to build, verify, and run it. This part also includes information on how to manually audit CapybaraNS's specification.
2. The source code for CapybaraKV and for the three unverified systems we compare against, and scripts to compile these systems and to verify CapybaraKV. It also includes instructions and scripts to run the experiments and produce the figures and tables in §6, as well as information on how manually audit CapybaraKV's specification.
3. Soundness proofs in Verus and Perennial that establish the correspondence between PoWER and CHL/crash invariants, and information on how to understand the proofs and verify that they hold. It does not include the formalization of the soundness proof in Perennial, but describes how to find that formalization in the Perennial repository and how to check it.

Hosting

The artifact is hosted publicly on GitHub at https://github.com/microsoft/verified-storage/tree/osdi25-artifact/osdi25 and on Zenodo at https://doi.org/10.5281/zenodo.15306499. The Zenodo artifact and the `osdi25` directory in the `osdi25-artifact` branch on GitHub have the same contents and contain the version of the artifact used during the artifact-evaluation process.

Requirements

Verifying and compiling CapybaraKV, CapybaraNS, and the soundness proofs requires no special hardware and should work on most platforms. The CapybaraKV performance experiments are only compatible with Linux platforms, as the unverified systems we compare against only support Linux. The experiments are compatible with both real Intel Optane DC PMM hardware and emulated PM on DRAM; however, at least 128 GB of Intel Optane DC persistent memory is required to reproduce the results presented in the paper. To reproduce the results in the paper, we also recommend at least 128 GB of DRAM and at least 32 cores. The README files included in each part of the artifact describe specific requirements and recommendations in more detail.

A.1 Installation

To install the artifact, either clone the GitHub repository and check out the `osdi25-artifact` branch, or download and unzip the Zenodo artifact. Follow the setup instructions in `osdi25/capybaraNS/README.md` to install dependencies for CapybaraNS. Follow the setup instructions in `osdi25/capybaraKV/README.md` to run a script that will install dependencies for CapybaraKV and build the systems used in performance experiments. Checking the proof of correspondence between PoWER and crash invariants is done automatically when verifying CapybaraKV (see below). The proof of correspondence between PoWER and Crash Hoare Logic is part of Perennial's source code and can be checked by building Perennial; see `osdi25/soundness_proofs/README.md` for instructions.

We include several "mini" versions of the full performance experiments described in §6 that take about 15 minutes to run. We suggest running these experiments after setting up CapybaraKV to check for issues before starting the longer-running experiments.

A.2 Experiment workflow

The files `osdi25/capybaraNS/README.md` and `osdi25/capybaraKV/README.md` describe the workflow to compile and verify each system, obtain line counts, manually audit their specifications, run performance experiments, and validate the claims made in the paper.

A.3 Expected results

CapybaraKV, CapybaraNS, and the soundness proofs are expected to verify successfully using their respective verification tools. The time it takes to verify each system can vary significantly depending on the hardware being used; see each system's README file for more information on expected verification times. In general, we expect that verifying CapybaraNS will take under a minute, and verifying CapybaraKV will take one to two minutes with one thread and between 15 and 40 seconds with 8 threads. We expect the lines of code outputted by the provided line-counting tools to exactly match those in Table 2.

When run in an environment similar to the experimental setup described in §6, we expect the PDFs and tables generated by scripts described in `osdi25/capybaraKV/README.md` to look similar to those presented in the paper. The default experiments provided by the artifact are smaller than those used in the paper, so we expect some variation in data collected from the artifact. Using different hardware (e.g., emulated PM or a smaller PM device) may result in noticeably different results. We describe expected differences in `osdi25/capybaraKV/README.md`.

References

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, April 2016.

[2] Piotr Balcer. Exploring the Software Ecosystem for Compute Express Link (CXL) Memory. https://pmem.io/blog/2023/05/exploring-the-software-ecosystem-for-compute-express-link-cxl-memory/, May 2023.

[3] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid PMem-DRAM key-value store. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, May 2021.

[4] BetrFS Developers. Checksumming. https://btrfs.readthedocs.io/en/latest/Checksumming.html, 2024.

[5] Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, and Clément Pit-Claudel. Practical verification of system-software components written in standard C. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 455–472, November 2024.

[6] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, 2019.

[7] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: A verified, concurrent, crash-safe journaling system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–439, July 2021.

[8] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association.

[9] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, October 2015.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, June 2010.

[11] Coq Development Team. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.8161141, June 2023.

[12] crc64fast Developers. crc64fast. https://crates.io/crates/crc64fast, 2024.

[13] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of Rust programs. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*, pages 90–105, October 2022.

[14] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 503–516, April 2021.

[15] Everspin Technologies, Inc. Spin-transfer Torque MRAM Technology. https://www.everspin.com/spin-transfer-torque-mram-technology.

[16] ext4 Developers. ext4 Data Structures and Algorithms. https://docs.kernel.org/filesystems/ext4/, 2024.

[17] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, pages 19–32, 1967.

[18] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 100–115, October 2021.

[19] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. Mumak: Efficient and black-box bug detection for persistent memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 734–750, May 2023.

[20] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 415–428, April 2021.

[21] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 99–115, November 2020.

[22] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, October 2015.

[23] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, October 2014.

[24] Hewlett Packard Enterprise. HPE Persistent Memory. https://www.hpe.com/us/en/servers/persistent-memory.html.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[26] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 429–442, April 2021.

[27] Tristan Hume. magic-trace. https://github.com/janestreet/magic-trace, 2024.

[28] Intel. Pmem-Redis. https://github.com/pmem/pmem-redis.

[29] Intel. Pmem-RocksDB. https://github.com/pmem/pmem-rocksdb.

[30] Intel Corporation. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[31] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv*, abs/1903.05714, August 2019.

[32] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–282, January 2011.

[33] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28(20):1–73, November 2018.

[34] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 45:1–32, January 2020.

[35] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic non-volatile memory crash consistency testing for full systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 933–950, July 2022.

[36] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 147–161, October 2019.

[37] KIOXIA America, Inc. XL-Flash. https://americas.kioxia.com/en-us/business/memory/xlflash.html, 2024.

[38] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):1–70, February 2014.

[39] Philip Koopman. CRC Polynomial Zoo. https://users.ece.cmu.edu/~koopman/crc/crc64.html, November 2019.

[40] Greg Kramer. Direct Drive - Azure's Next-generation Block Storage Architecture. https://www.sniadeveloper.org/events/agenda/session/347, 2022. SNIA Developer Conference.

[41] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994.

[42] Leslie Lamport and Stephan Merz. Prophecy made simple. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2):1–27, April 2022.

[43] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 438–454, November 2024.

[44] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 718–733, May 2023.

[45] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 348–370, April 2010.

[46] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1187–1202, March 2020.

[47] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–425, April 2019.

[48] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey and ACE: Systematically testing file-system crash consistency. *ACM Transactions on Storage*, 15(2):1–34, April 2019.

[49] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, February 2019.

[50] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1047–1064, November 2020.

[51] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 252–269, October 2017.

[52] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, October 2014.

[53] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, July 2002.

[54] Andy Rudoff. Persistent memory programming. *;login:*, 42(2):34–40, 2017.

[55] Andy Rudoff. Re: 8 byte atomicity & larger store operations. https://groups.google.com/u/1/g/pmem/c/6_5daOuEI00/m/nY_mtKd0CAAJ, October 2020.

[56] Rust Developers. Type Layout. https://doc.rust-lang.org/reference/type-layout.html, 2024.

[57] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–16, November 2016.

[58] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring data integrity in storage: techniques and applications. In *Proceedings of the ACM Workshop on Storage Security and Survivability*, pages 26–36, November 2005.

[59] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: verifying liveness of cluster management controllers. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 649–666, July 2024.

[60] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. Modular reasoning about separation of concurrent data structures. In *Proceedings of the 22nd European Symposium on Programming (ESOP)*, pages 169–188, Rome, Italy, March 2013.

[61] Joseph Tassarotti et al. Perennial asynchronous disk prophecy equivalence proof. https://github.com/mit-pdos/perennial/blob/master/src/goose_lang/ffi/async_disk_equiv.v, November 2021.

[62] Joseph Tassarotti et al. Perennial asynchronous disk prophecy model. https://github.com/mit-pdos/perennial/blob/master/src/goose_lang/ffi/async_disk_proph.v, November 2021.

[63] Tom's Hardware. Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift, August 2022.

[64] Doug Voigt. Persistent memory atomics and transactions. https://www.snia.org/educational-library/persistent-memory-atomics-and-transactions-2017, October 2017.

[65] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. Modular specification and verification of closures in Rust. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–29, October 2021.

[66] XFS Developers. XFS Self Describing Metadata. https://www.kernel.org/doc/Documentation/filesystems/xfs-self-describing-metadata.txt, 2024.

[67] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 478–496, October 2017.

[68] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, February 2020.

[69] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, December 2004.



Fast and Synchronous Crash Consistency with Metadata Write-Once File System

Yanqi Pan[†], *Wen Xia*[†]✉, *Yifeng Zhang*[†], *Xiangyu Zou*[†], *Hao Huang*[†], *Zhenhua Li*[‡], *Chentao Wu*[¶]
[†]*Harbin Institute of Technology, Shenzhen*
[‡]*Tsinghua University* [¶]*Shanghai Jiao Tong University*
✉ *Corresponding Author: Wen Xia (xiawen@hit.edu.cn)*

Abstract

Low-latency persistent memory (PM) encourages file systems to pursue synchronous crash consistency. However, existing crash consistency approaches, such as *journaling* and *log structure file system*, incur many small, random, and ordered metadata I/Os, failing to exploit PM I/O potential. We propose a new file system model called *metadata write-once file system (WOFS)* to achieve fast and synchronous crash consistency. The key idea is to generate specific metadata for each file operation as a checksum-protected package and write it once with a single ordering point. The package is then managed to provide file abstractions through a package translation layer without extra writes. Using an array of techniques to generate, organize, and recover from packages, WOFS can provide practical, efficient, and reliable file system services. We implement WOLVES as a WOFS prototype in Linux kernel. Experiments using benchmarks and applications suggest that WOLVES can recover from crashes, improve operation throughput, and potentially reach PM I/O bandwidth limits.

1 Introduction

Persistent Memory (PM) technologies, such as 3D-XPoint [1, 2] and CXL-SSD [3–5], have shown the merits to supercharge data-related scenarios [6–9]. Specifically, PM can be attached to the memory bus through specific protocols and accessed via CPU loads/stores without the block layer [4, 10–12].

While conventional block-based file systems require orchestrating I/O orders across many layers and using expensive disk flushes for durability [13–15], PM file systems enable fast synchronous I/Os by waiting for I/Os to reaching PM interfaces without intermediate layers. This encourages PM file systems to pursue synchronous crash consistency [16–29], which ensures file operations are immediately durable without explicit synchronization (*e.g.,* `fsync`), thereby promising to greatly simplify application crash consistency efforts [30–33].

Existing PM file systems follow two existing methodologies to achieve synchronous crash consistency. The first kind advocates **additional writes** for simplicity [34]. *Journaling file system (JFS)* is a representative approach [16,17,20,22,23, 35, 36], which reserves a contiguous PM area as the journal. JFS conducts an ordered metadata write to journal followed by an ordered commit block, and then issues multiple orderless in-place updates to file system metadata. As a result, system failures can be fixed by replaying metadata update backups from transactions. However, journaling with redundant metadata I/Os, even with optimizations such as transactional checksum [15, 20], reduces PM throughput dramatically.

The second kind carefully orders I/O sequences **without additional writes**. Approaches such as *log-structured file system* [21, 28] and *synchronous soft update* [25] enforce update ordering on different file system metadata (*e.g.,* inode, dentry, etc.). For instance, creating a file in SquirrelFS [25] involves first allocating and initializing a new inode, and then filling in a dentry to point to the inode. These ordering points [37] ensure that crash points are reasoned and can be fixed for recovery, thereby eliminating additional metadata writes; however, they unfortunately incur many small, random, and ordered metadata I/Os that hinder PM performance [10, 11].

Our motivational experiments on three representative PM file systems with different crash consistency techniques show that these metadata I/Os (for crash consistency) dominate more than 70% of the total I/O time, causing a severe PM bandwidth waste. We thus propose a new file system model, *metadata write-once file system (WOFS)*, which aims to minimize metadata I/Os and ordering points for fast and synchronous crash consistency. The key idea is to generate specific metadata for a file operation as a checksum-protected package and write it once with a single ordering point (*i.e.,* one synchronous I/O). Consequently, WOFS ensures that every metadata I/O (or package) is atomic and durable, and that WOFS can rely on replaying them for reliable recovery.

However, designing a file system that manages metadata as packages is non-trivial. We thus present a range of techniques: (1) to generate packages for various file system operations in Linux, we abstract packages for basic operations and use them to build complex operations; (2) to parse packages to provide metadata objects (*e.g.,* inode) for upper file/dir abstractions, we propose a *package translation layer* to gather fields from written packages; (3) to organize packages/data blocks in PM, we deploy a non-log layout and use reallocation to reclaim invalidated space; (4) to recover from packages after crashes, we propose *coarse persistence* to enable quick location of a group of packages for fast recovery. Consequently, these techniques make WOFS practical, efficient, and reliable.

We built WOLVES as a WOFS prototype on Intel PM. By applying two other techniques tailored to PM, *i.e., huge allocation* and *read ahead*, WOLVES further improves sequential I/O throughput. Finally, we port WOLVES to another PM platform (*i.e.,* an emulated memory-semantic SSD platform [3]) to show the generality of WOFS. Experiments

show that WOLVES can achieve 97.3%–99.1% PM I/O bandwidth under several write-intensive benchmarks. Using real-world RocksDB, we further show that WOLVES achieves 1.20–6.73× throughput compared to existing PM file systems.

In a nutshell, the contributions of this paper are three-fold:

- We make an in-depth analysis of existing crash consistency design principles atop PM (§2) and reveal that their many small, random, and ordered metadata I/Os squander fast PM bandwidth, and become bottlenecks for file operations (§3).
- We propose a WOFS model for fast and strong crash consistency. WOFS generates specific metadata for each file operation as a checksum-protected package and writes it once with a single ordering point. A series of techniques are proposed to generate, organize, and recover from packages, making WOFS practical, efficient, and reliable (§4).
- We have implemented the WOLVES file system to realize the efficiency of WOFS (§5). Experiments show that WOLVES can reliably recover from system crashes, outperform existing PM file systems across a range of workloads, and potentially reach PM I/O bandwidth upper limits (§6).

2 Background

2.1 Fast and Synchronous PM I/O Stack

PM characteristics. PM offers performance comparable to DRAM and data persistence similar to storage [1, 2, 4, 38, 39]. PM can be accessed via CPU load/store instructions through specific protocols (*e.g.,* DDR-T [10]). Compared to traditional HDDs/SSDs, PM is superior due to two unique characteristics: (1) *byte-addressability* that enables file systems to use direct access to bypass the block layer, and (2) a *persistent buffer* (*e.g.,* XPBuffer in Intel PM [10]) to ensure immediate durability when I/Os reach the PM interface [11, 40].

Fast synchronous PM I/O stack. The above PM features enable fast synchronous I/Os [25]: for each issued I/O request, PM file systems only require issuing a `PCOMMIT` (*i.e.,* CPU flush plus a memory barrier, such as `clwb+sfence`) to wait for the I/O to reach the persistent buffer (*e.g.,* 50–300 ns [11]). In contrast, traditional block-based file systems require waiting for data transfer across multiple layers to the media [13, 15], which is time-consuming (*e.g.,* tens of microseconds [41]).

2.2 Synchronous Crash Consistency

With its fast synchronous I/Os, PM file systems pursue synchronous crash consistency [16–29]: each file operation is durable after its return without synchronization. This contrasts with the optimistic approach for slow disks [13, 15, 37], which allows operation loss for higher I/O throughput. Achieving synchronous crash consistency is important and charming as it can simplify application crash consistency efforts, especially for databases that suffer incorrect `fsync` behaviors [31, 32] and the strict NFS protocol that issues `fsync` for every operation [30]. Existing PM file systems follow two conventional methodologies for synchronous crash consistency, as below.

Tech.	I/O order (*e.g.,* `write`)	MOrd	MIOs	GC
JFS [22, 36]	$D \rightarrow J_M \rightarrow J_C \rightarrow M$[a]	3	> 3	Reuse
CK [20]	$D \rightarrow \overline{J_M \vert J_C} \rightarrow M$	2	> 2	Reuse
LFS [21]	$D \rightarrow (GC) \rightarrow m_e \rightarrow m_t$	≥ 2	≥ 2	Copy
SSU [25]	$D \rightarrow m_1 \rightarrow m_2 \cdots m_N$	N	N	Reuse
WOFS	$D \rightarrow \overline{J_M \vert J_C} \vert (m_{cp})$[b]	1	~ 1	Reuse

[a]: M is short for $m_1 \vert m_2 \vert \cdots \vert m_N$, where m_i is one metadata update.
[b]: m_{cp} denotes coarse bitmap persistence used for fast recovery (§4.5).

Table 1: Crash consistency techniques in PM file systems. *MOrd and MIOs denote the number of metadata ordering points and metadata I/Os. The line below CK separates techniques w/ and w/o additional writes. CK and SSU denote transactional checksum and synchronous soft update. WOFS is our proposed crash consistency method. The parentheses, e.g., (GC), indicate that the I/O does not always occur. The Copy-GC indicates that the file systems rely on data migration to reclaim invalidated space, while Reuse-GC indicates that the invalidated space is simply reallocated.*

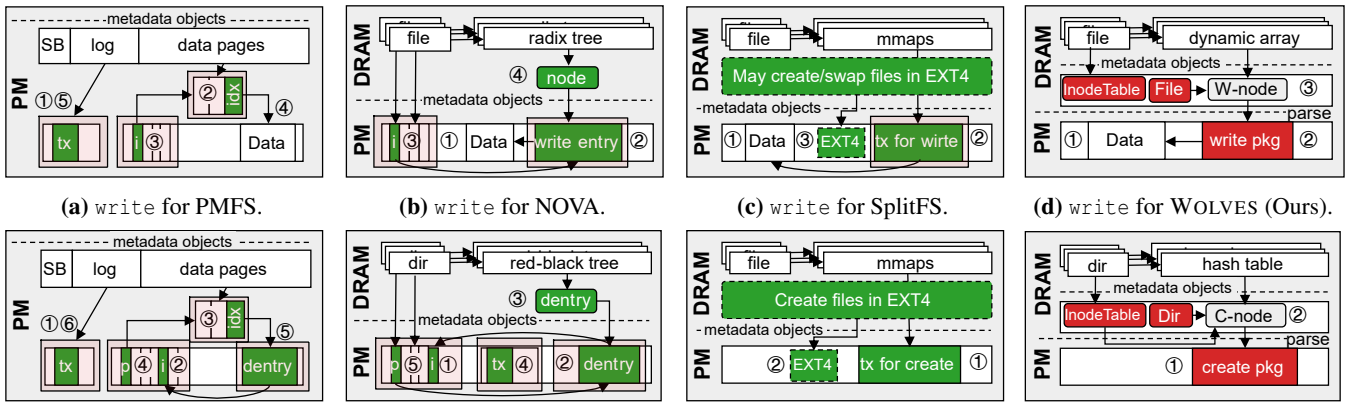
Synchronous crash consistency with additional writes. These approaches back up metadata before updating them, which are simple but incur redundant writes. The most representative one is *journaling file system (JFS)*. JFS first writes data blocks (D), then wraps metadata updates as a transaction and writes it to a journal area (J_M). Afterward, JFS writes a commit flag (J_C) atomically to the journal. Finally, JFS conducts orderless metadata updates in place (M). This entails an I/O order of $D \rightarrow J_M \rightarrow J_C \rightarrow M$[1]. As a result, JFS can recover by checking and replaying transactions after crashes [13, 15]. Many PM file systems [22, 23, 36] leverage journaling for crash consistency.

To optimize JFS I/O, *transactional checksum (CK)* [42] is proposed. Specifically, CK writes J_M, J_C, and their checksum together without ordering (*i.e.,* $\overline{J_M \vert J_C}$[2]). As a result, incomplete writes can be verified by recalculating the checksum of J_M and J_C and comparing it with the stored one. Thus, CK has an I/O order of $D \rightarrow \overline{J_M \vert J_C} \rightarrow M$. Recent SplitFS [20] is a representative that leverages CK to accelerate journaling.

Synchronous crash consistency without additional writes. These approaches carefully order metadata updates to eliminate additional metadata writes. *Log-structured file system (LFS)* is the most representative one used in PM file systems, such as NOVA [21]. Specifically, LFS conducts metadata writes in an append manner [21, 43–45]. LFS regards the storage as a log and appends file/dir metadata to the log as a log entry (m_e); it then maintains *log tail* (m_t) to preserve the current write pointer. As a result, LFS can recover by scanning and checking log entries until the log tail. To accommodate the ever-growing log layout, LFS leverages the copy-based garbage collection (GC) to compact the valid log entries. As a result, the I/O order of LFS is $D \rightarrow (GC) \rightarrow m_e \rightarrow m_t$, where the parentheses indicate that the GC is not always triggered.

Another approach is synchronous soft update (SSU) proposed by SquirrelFS [25]. SSU is similar to LFS that performs

[1] $A \rightarrow B$ indicates that the persistence of B occurs after that of A.
[2] The overline indicates that J_M and J_C can be written concurrently.



(a) `write` for PMFS. **(b)** `write` for NOVA. **(c)** `write` for SplitFS. **(d)** `write` for WOLVES (Ours).

(e) `create` for PMFS. **(f)** `create` for NOVA. **(g)** `create` for SplitFS. **(h)** `create` for WOLVES (Ours).

Figure 1: Analysis of I/O ordering on write and create of existing PM file systems. *[i] and [p] are inode and parent inode; [tx] is transaction. The light-red squares indicate disaggregated in-PM metadata I/Os. The circled numbers denote update orders. PMFS, SplitFS, and NOVA represent JFS, JFS with transactional checksum, and LFS. Our WOLVES (§5) writes metadata once with a single ordering point.*

ordered updates on multiple different metadata objects (*e.g.,* inode, m_1, dentry, m_2, etc.) to satisfy three basic rules [34]: (1) never point to a structure before it has been initialized; (2) never re-use a resource before nullifying all previous pointers to it; and (3) never reset the old pointer to a live resource before the new pointer has been set. With these rules, SSU ensures that incomplete writes will never be visible to the file system, thus achieving crash consistency. Consequently, these rules form an I/O order as $D \rightarrow m_1 \rightarrow m_2 \cdots m_N$.

Summary. Table 1 summarizes existing crash consistency techniques, which suggests that these techniques incur multiple metadata I/Os and ordering points in critical paths.

3 Observations and Motivations

Nevertheless, unorchestrated I/Os and ordering points are essentially deficient in PM: (1) the coarse flush unit of persistent buffers (*e.g.,* 256-byte XPLine [10]) mismatches with small metadata, and thus can introduce I/O amplifications; (2) ordering points lead to wait for previous I/O operations, reducing I/O parallelism. Thus, we ask: *what exacts the toll on metadata I/Os in crash consistency processing for PM?* To answer this, we make an in-depth analysis of three PM file systems.

3.1 Analyzing I/O Path of PM File Systems

We analyze PMFS and SplitFS for JFS and CK with additional writes, and NOVA for LFS without additional writes. We omit SSU as it is similar to LFS, and we report SquirrelFS in §6.11.

PMFS I/O path analysis. PMFS is a classic JFS in PM [22]. It reserves a log space at the start of the PM for journaling, and the remaining space is organized as a B-tree for storing inodes and data blocks. For `write`, as Figure 1a shows, ① PMFS wraps metadata updates as a transaction (J_M), ② allocates blocks by traversing the B-tree and ③ updates the inode (M). Finally, ④ PMFS persists data (D) and ⑤ writes a commit entry to the log (J_C). The `create` is similar to `write` but needs to create a new inode (see ② in Figure 1e). Note that PMFS changes the original JFS order to $J_M \rightarrow M \rightarrow D \rightarrow J_C$, but its metadata I/Os and ordering points remain the same.

Workload	Avg. fsize	R/W size	R:W	#. files
FIO-SW	32GiB	0KiB/4KiB	0:1	1
FIO-RW	32GiB	0KiB/4KiB	0:1	1
Fileserver	128KiB	4KiB/4KiB	1:2	10K
Varmail	32KiB	1MiB/16KiB	1:1	10K
Webserver	64KiB	1MiB/8KiB	10:1	1K
Webproxy	32KiB	1MiB/16KiB	5:1	10K

Table 2: Characteristics of the evaluated six workloads. *The first two FIO workloads perform write operations. The last four Filebench workloads perform various open/create/write/read/-close/delete/fsync file operations to emulate real-world scenarios.*

NOVA I/O path analysis. NOVA [21] is the first LFS designed for PM. It associates each inode with a linked metadata log and leverages DRAM for fast indexing. For `write`, as Figure 1b shows, NOVA ① persists the data (D), ② appends a write entry to its inode log (m_e), ③ updates the log tail in the inode (m_t), ④ creates a DRAM node to index the entry. For multi-inode operations, such as `create`, NOVA introduces journals to protect the atomicity of affected log tails. As Figure 1f shows, `create` involves journal writes (④), inode updates (① and ⑤), and dentry appends (②). NOVA triggers GC whenever the log is full and needs to be extended.

SplitFS I/O path analysis. SplitFS is a user-space file system that utilizes transactional checksums for fast journaling [20]. It further accelerates data writes (D) via `mmap` provided by underlying EXT4, and EXT4 assists its metadata updates (M), as shown in Figures 1c (③) and 1g (②).

3.2 Analyzing Crash Consistency Overheads

Methodology. We break down the I/O path of PMFS, SplitFS, and NOVA (by inserting timing code around I/O operations) to

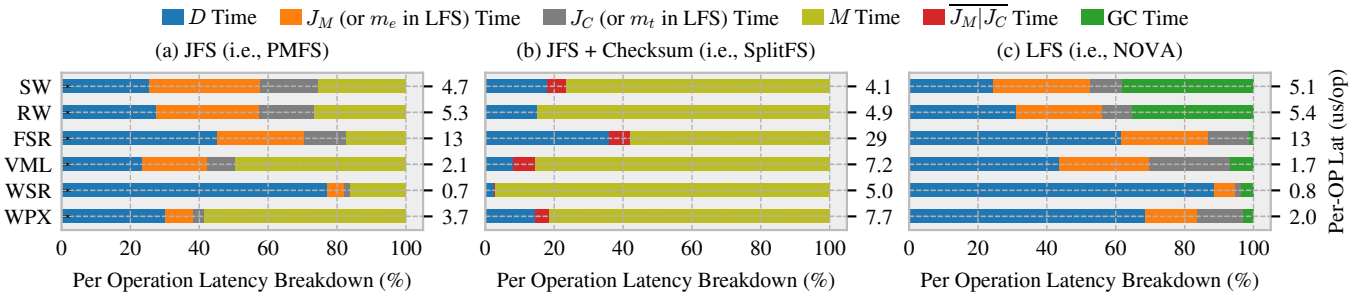


Figure 2: The study of crash consistency techniques in PM. *The right y-axis presents the average per-operation latency. Note that in RW and WSR, SplitFS does not write transactions (i.e., $\overline{J_M|J_C}$=0). SW and RW are FIO sequential/random 32 GiB write with 4 KiB per I/O and sync engine. FSR, VML, WSR, and WPX are short for Filebench workloads: fileserver, varmail, webserver, and webproxy. Note that NOVA's GC is significant in FIO workloads as it triggers GC upon log extensions [21]; consequently, larger files trigger more frequent GC, incurring higher overhead. In comparison, in Filebench workloads where files are small, GC overhead remains low.*

learn the overheads. We run six widely-used single-threaded micro- and macro-benchmarks [21, 22] as listed in Table 2.

Observation. Figure 2 suggests that metadata I/O time for synchronous crash consistency can even exceed the data I/O time. Specifically, PMFS, SplitFS, and NOVA spend 22.9%–76.5%, 63.8%–97.4%, and 11.3%–75.5% of the total I/O time on metadata in the six evaluated workloads. We find that these metadata overheads come from three main aspects:

- *Random metadata I/Os lead to persistent buffer hit misses, which causes I/O amplifications*. We study I/O amplifications of PMFS in the SW workload. Using *ipmctl* [46], we find that metadata I/Os of PMFS are increased from a theoretical ∼2.9 GiB (by calculating all issued metadata I/Os) to a measured ∼8.0 GiB, leading to 2.8× amplifications.
- *Ordered metadata I/Os lead to waiting for previous data transfer and limit PM I/O concurrency*. As Figure 2a and 2b show, PMFS spends 6.73%–49.2% metadata I/O time on transaction write and commit (steps ① and ⑤ in Figure 1a), while the overheads are reduced to 3.99%–6.27% in SplitFS with transactional checksum that eliminates ordering.
- *Increasing metadata I/Os further exacerbate the overheads*. For NOVA, the GC exacerbates metadata I/O numbers, causing an average of ∼13 metadata accesses per block I/O. This, as shown in Figure 2c, leads to ∼38.0% GC overheads. Meanwhile, metadata I/Os are increased from a theoretical ∼4.2 GiB to a measured ∼11.1 GiB (2.8×).

Significance. In a nutshell, many small, random, and ordered metadata I/Os lead to severe overheads. Consequently, in SW, existing PM file systems achieve less than 50% (not shown in the figure) of PM write bandwidth (∼2.26 GiB/s on our machine). This observation significantly challenges existing crash consistency techniques under the synchronous context.

3.3 Motivation

Intuitively, one solution to the above deficiency is to minimize metadata I/Os and ordering points for crash consistency.

Attempts. While many existing crash consistency designs attempt to achieve this goal, they ultimately fall short. Particularly, (1) transactional checksum achieves one $\overline{J_M|J_C}$ transaction write, but it fails to eliminate multiple, random, orderless metadata updates; (2) LFS successfully eliminates redundant journal writes via ordered log entry writes (m_e) and tail commit (m_t). However, further attempts to eliminate *GC* are failed as LFS must use a copy-based GC to compact the layout; (3) although SSU does not have to trigger GC [25], it still suffers from small, random, and ordered metadata I/Os as existing file systems manage multiple metadata objects (*e.g.,* inode) for file/directory abstractions, leading to significant I/O overheads (*i.e.,* ∼30% degradation as shown in Figure 15 in §6.11).

Insight. We note that existing crash consistency techniques struggle to *orchestrate and optimize I/Os across a range of metadata objects* (*e.g.,* inode, log entry/tail, etc.). This methodology, however, misaligns with the goal of minimal metadata I/Os and ordering points. Consequently, we are motivated to rethink file system metadata itself. Intuitively, by designing specific metadata for each file operation (similar to J_M) and attaching a checksum header to protect it (similar to J_C), the file system can write the metadata (*i.e.,* $m_{op} = \overline{J_M|J_C}$) once with a single ordering point: $D \rightarrow \overline{J_M|J_C}$.

Challenges. However, designing metadata for each operation is non-trivial, as reflected in four key questions: (1) How to practically design m_{op} considering various Linux file operations [47, 48]? (2) How to manage m_{op} to be compatible with traditional file/dir abstractions (*e.g.,* inode)? (3) How to organize m_{op} in PM efficiently? (4) How to recover from m_{op} reliably? We answer these questions in the next Section.

4 Metadata Write-Once File System Model

Motivated by §3, we propose *metadata write-once file system (WOFS)*. The key idea of WOFS is to generate specific, aggregated metadata for each file operation as a checksum-protected package (corresponding to the m_{op} described in §3.3), and write the package once with a single ordering point.

4.1 Overview and Design Principles

The WOFS overview is shown in Figure 3, as described below.

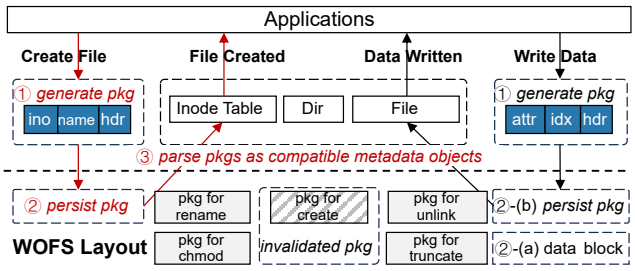


Figure 3: WOFS overview. *Each file operation corresponds to a package that contains necessary metadata fields, which can be written once. The packages are then parsed for file/dir abstractions.*

Package overview. In WOFS, a generated package contains necessary fields (denoted as J_M) for a file operation (*e.g.,* name for `create`). Further, a header is reserved for each package as a commit flag (denoted as J_C), which stores a *magic number*, a *type* for package type, a *timestamp*, and a CRC32 checksum[3] for package write once (*i.e.,* $\overline{J_M|J_C}$), as shown in Listing 1.

Workflow overview. As illustrated in Figure 3, for metadata operations such as `create`, WOFS ① generates a package for `create`, ② writes it once using a `PCOMMIT`, *i.e.,* $\overline{J_M|J_C}$, and ③ parses it to organize inode, inode table, and directory hierarchy. Data operations (*e.g.,* `write`) have a different I/O mechanism: WOFS first persists data (②-(a)) and then writes the package that references them (②-(b)), *i.e.,* $D \rightarrow \overline{J_M|J_C}$.

Crash consistency overview. With packages and the metadata write-once scheme, WOFS can achieve fast and synchronous crash consistency with minimal metadata I/Os and ordering points. First, metadata operations involve only $\overline{J_M|J_C}$; therefore, once a failure occurs, WOFS can detect incomplete packages by checking the checksum of J_M and J_C, and discard invalid ones during recovery. Second, data operations follow $D \rightarrow \overline{J_M|J_C}$; therefore, a crash before or during the $\overline{J_M|J_C}$ leads to an incomplete package, which can be discarded; meanwhile, as data blocks are not referenced by any valid package, they can be regarded as free space without consistency issues.

Design principles. The overall workflow and crash consistency principles of WOFS seem simple, but it requires further design considerations regarding packages, as guided below.

- *Generate minimal packages*. It is partially unnecessary to generate the specific package for each file operation as many operations share a similar semantic [47, 48]. Thus, WOFS aims to generate minimal packages that can represent or be combined to represent file operations (§4.2).
- *Parse packages to provide compatible metadata objects*. As packages upend the traditional file system metadata objects (*e.g.,* inode), WOFS aims to provide compatible metadata objects for efficient file/directory abstractions (§4.3).
- *Organize packages and data blocks in a non-log manner*. WOFS aims to allocate packages/data blocks across PM rather than in a log manner to avoid copy-based GC (§4.4).

[3]CRC32 is widely used for protecting file systems' metadata [20–22, 36, 45].

```
hdr = pkg->hdr;
hdr->type = WRITE; hdr->ts = current_time();
// fill other attributes inside pkg ...
hdr->CRC32 = CRC32(pkg, pkg->size); // checksum
// write package once with a single ordering point
PCOMMIT(pkg, pkg->size);  // clwb+sfence
```

Listing 1: Package write once I/O scheme. *WOFS uses the transactional checksum scheme [20, 42] to persist the package with a single ordering point (i.e., one `PCOMMIT`).*

- *Recover from packages after crashes efficiently*. The non-log layout complicates failure recovery as WOFS stores packages across the whole PM. WOFS aims to provide a coarse bitmap to quickly locate and check packages without frequent random I/Os to support fast recovery (§4.5).

4.2 WOFS Package Design

Considering 15+ file system operations in modern Linux [47, 48], it is partially unnecessary to generate the specific package for each file operation as many operations have similar semantics, which also increases complexity and reduces flexibility. Our goal is to abstract the most fundamental file operations, similar to CRUD operations in database [49].

Generate atomic package. We observe that file system operations can be categorized into CRUD: create (*e.g.,* `create`), read (*e.g.,* `read`, `lstat`), update (*e.g.,* `write`, `chown`), and delete (*e.g.,* `rm`). As read operations do not alter file system states, we omit read packages. For update operations, we find that they can be further divided into data (*e.g.,* `write`) and metadata modification (*e.g.,* `chown`). Therefore, we abstract the following four packages as the *atomic package*.

- *Create pkg* (256 bytes) for the operations that create new inodes, such as `create` and `link`. It contains 64 byte static file attributes, *e.g.,* create time and inode number (ino). The next 128 bytes store the parent ino, the linked ino, and the name entry. The remaining 64 bytes store the attributes of the parent inode (*e.g.,* link changes) and the package header.
- *Write pkg* (64 bytes) for the operations that allocate new data blocks, such as `write` and `fallocate`. It uses the extent [14, 21] to index the continuously allocated data blocks and records the time and size changes of the inode.
- *Attr pkg* (64 bytes) for the operations that change the attributes (*e.g.,* mode, uid, etc.) of an inode.
- *Unlink pkg* (64 bytes) for the operations that reduce the link of an inode. It contains the changed attributes of both parent (*e.g.,* time and size) and unlinked inode (*e.g.,* link counts and the inode number that is to be unlinked).

Generate compound package with crash consistency. We use multiple atomic sub-packages to represent complex file operations, *e.g.,* `rename` (=`link`+`unlink`). To ensure the crash consistency of compound packages, we reserve a *forward-pointer* in the package header to link sub-packages. We assign sub-packages an extra compound operation type to distinguish them from normal ones. For example, we assign

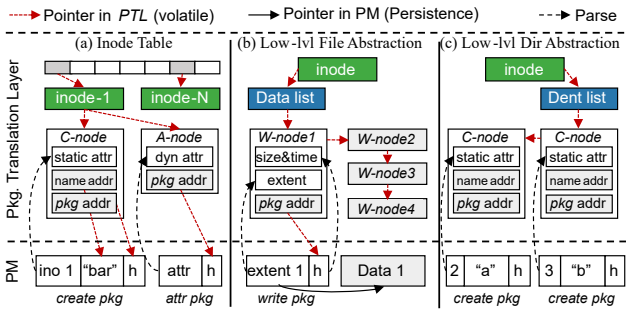


Figure 4: Package translation layer. *h is package header. PTL parses packages as pkg-nodes (*e.g., *C-node/A-node/W-node for the create/attr/write pkg) and organizes different pkg-nodes to provide compatible metadata objects for file/dir abstractions (§5.2).*

the type `CREATE|RENAME` to the *create pkg* used in a rename operation. Consequently, WOFS can write sub-packages without ordering points, *i.e.,* $P_1|P_2|...|P_N$, where P_i is short for a sub-package I/O (*i.e.,* $J_M|J_C$) and P_i points to P_{i+1}.

We run `rename` as an example. `rename` can be regarded as first hard linking a new file to the old one and then unlinking the old one [37]; therefore, it can be represented using *create pkg* (P_{create}) and *unlink pkg* (P_{unlink}). We now show that WOFS can safely issue $P_{create}|P_{unlink}$: If only P_{unlink} is partially written, WOFS can detect the incompleteness by following the *forward-pointer* in P_{create} and then discard `rename`. If only P_{create} is partially written, WOFS detects the incompleteness by determining that there is a P_{unlink} with *rename* type, but no P_{create} pointing to it. Note that using a *dual-pointer* can simplify this reverse detection, but it would misalign the atomic packages from the 64-byte cache line, which is currently avoided in our design. Finally, if both of them are partially written, WOFS cannot detect valid sub-packages, which is equivalent to the case where `rename` has never happened.

4.3 Package Translation Layer

By aggregating metadata of an operation as a package, WOFS upends the traditional metadata objects (*e.g.,* inode, dentry, etc.). Consequently, WOFS should parse packages to provide compatible file/directory abstractions. We introduce *package translation layer (PTL)* to achieve this goal. The basic management unit in PTL is *pkg-node*, which contains the parsed data from the package for fast access and the address of the package for reclamation (§4.4). These nodes are organized to form the compatible metadata objects, as shown in Figure 4.

- *Inode table*. To construct an inode, PTL combines the pkg-nodes of *create pkg* and *attr pkg* to represent its static and variable attributes. PTL maps the inode number to the inode as the inode table for fast search. An *inode manager* is introduced to allocate/free the inode number.
- *Low-level file abstraction*. PTL is responsible for locating data blocks to provide low-level file abstraction. As Figure 4b shows, PTL associates a list of *W-node*s for each regular file (*i.e., data list*), where the *W-node* contains the extent structure parsed from the *write pkg*. The upper file abstraction can reorganize these nodes for fast data access.
- *Low-level directory abstraction*. Similarly, PTL maintains a list of *C-node*s (*i.e., dent list*) to locate the sub-inodes under a directory. These nodes are further organized by the upper directory abstraction to provide high search performance (*e.g.,* using a hash table for name searching).

4.4 Non-log Layout and Package Reclamation

WOFS introduces a non-log layout that allocates blocks/packages across PM similar to *malloc*, which has two benefits: (1) allocate space anywhere to exploit PM fast I/O and erase-free nature [10]; (2) reclaim space via reusing (instead of heavy copy-based GC), which reasons invalidations by determining the causal-order of operations in the critical path, and reallocates invalidated space (similar to *free*), as follows.

- *Create pkg*. The deletion of a file (by writing an *unlink pkg*) invalidates its *create pkg*. After that, the space of *create pkg* recorded in the *C-node* (*i.e., pkg addr*) is marked as free and can be reallocated for new packages.
- *Write pkg*. Size change (*e.g.,* `truncate`) leads to the invalidation of the *write pkg* and corresponding data blocks. Overwrite might also lead to invalidation but it depends on the implementation. For example, if the file system adopts Copy-on-Write (COW) for data atomicity, overwrite will invalidate the old *write pkg* accordingly (see details in §5.2).
- *Attr pkg*. Any attribute changes result in the persistence of a new *attr pkg* of the file, which leads to the invalidation of the old *attr pkg* that belongs to the same file.
- *Unlink pkg*. WOFS checks whether to invalidate an *unlink pkg* during the creation of a file. This is because the *unlink pkg* cannot be reclaimed until its pointing *create pkg* is reallocated and overwritten, or WOFS might find a stale but valid *create pkg* during the recovery. Specifically, WOFS maintains an *order map* in the PTL, which maps the address of *create pkg* to that of *unlink pkg*. Whenever a *create pkg* is durable, WOFS looks up its written address in the *order map* to reclaim the associated *unlink pkg*.
- *Compound package* cannot be reclaimed until all the atomic packages are invalidated. To track this, we extend PTL by generating each compound package a *comp-pkg-node* and adding a reverse mapping from the *pkg-node* to the *comp-pkg-node* to notify the reclaiming of the atomic packages.

4.5 Fast Recovery with Coarse Persistence

For recovery, WOFS requires locating packages to check consistency and rebuilding PTL to manage packages. However, it is non-trivial to quickly locate packages as they are distributed across the PM due to the non-log layout design. We thereby first explore package locating approaches.

Locate packages with dump-restore. A naïve approach is *dump-restore (DR)* [21]: WOFS dumps the addresses of packages in a bitmap (where each bit for a 64-byte slot, as packages are 64-byte aligned) at the unmount time. During recovery,

WOFS uses this bitmap to locate and parse the packages. However, this approach still causes a PM scan after the system crashes due to the failure to timely dump the bitmap. *DR* can be optimized (*i.e., DR-OPT*) by "tagging" unallocated blocks during format, allowing them to be skipped during recovery, but it cannot avoid scanning allocated data blocks.

Locate package with coarse persistence. We argue that a centralized structure is a must for fast locating, but its impact can be minimized. We propose *Coarse Persistence (CP)*, which allocates a coarse *pkg-group* (*e.g.,* 4 KiB) to hold multiple packages and timely persists the address of *pkg-group* into the bitmap once it is allocated (*i.e.,* a bit for a group). Therefore, WOFS can scan the bitmap to locate the *pkg-group*, then probe to check valid packages, and finally rebuild PTL.

To ensure the crash consistency of CP, a straightforward approach is to insert an ordering point between bitmap update (m_{cp}) and the package I/O ($P = \overline{J_M|J_C}$), *i.e.,* $D \rightarrow m_{cp} \rightarrow P$. However, this ordering is unnecessary (*i.e.,* $P|m_{cp}$). Particularly, (1) if m_{cp} is durable, P can be detected through the bitmap and checked for completeness; (2) Otherwise, P cannot be accessed through the bitmap and is simply discarded. This is safe as m_{cp} occurs only when a new *pkg-group* is allocated; therefore, discarding P is equivalent to handling a normal data operation crash as described in §4.1.

Recovery with coarse persistence. After locating packages using CP, WOFS fixes inconsistencies and re-parses packages to rebuild the PTL. To fix inconsistencies, WOFS must identify and reclaim stale packages. It does so by determining the causal order between packages, treating logically reclaimed packages as invalidated, *e.g.,* a newer *unlink pkg* invalidates its corresponding older *create pkg*. Specifically, WOFS first detects successfully written packages using the magic number and checksum, then tracks their versions using timestamps in headers, and finally determines their causal order based on operation semantics (following rules similar to those in §4.4). Compound packages are checked and fixed using the forward-pointer mechanism, as described in §4.2.

Crash safety semantics. Consequently, WOFS's crash safety guarantees that packages preserve the same *causal order* as before the crash, rather than their *issue order*. For example, assume Thread-1 writes a *create pkg* C_1 for a file, followed by a *write pkg* W_1, and Thread-2 writes a *write pkg* W_2 to another file. These writes are issued in the order $C_1(t_1)$, $W_2(t_2)$, and $W_1(t_3)$, where t_i denotes the timestamp and $t_1<t_2<t_3$. A crash may leave W_2 partially written but C_1 and W_1 fully written; in this case, WOFS treats $C_1(t_1)$ and $W_1(t_3)$ as valid to preserve the correct causal order. On the other hand, WOFS does not require $W_2(t_2)$ for recovery since W_2 has no causal dependency on W_1 (*i.e.,* they operate on different files).

5 WOFS Implementation

We built WOLVES as a WOFS prototype in Linux kernel 5.1.0 with 12,000+ lines of C code. We also introduce several techniques to optimize I/O performance by leveraging Intel PM characteristics (*e.g.,* XPLine size and memory interface).

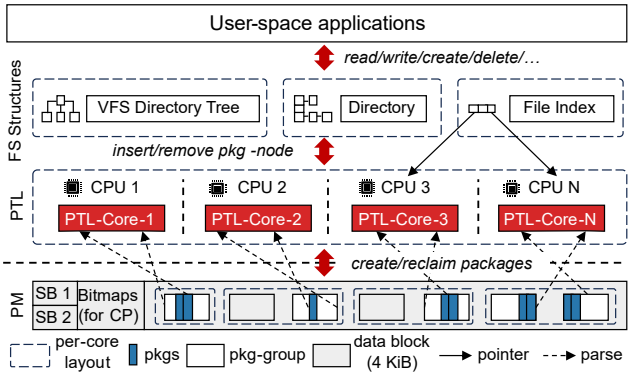


Figure 5: WOLVES file system architecture. *Packages are allocated within pkg-groups for fast recovery. WOLVES then writes packages with one* `PCOMMIT`. *PTL is then built to manage packages as traditional metadata objects for upper file/dir abstractions. To enable multi-core scalability, PTL is deployed in a per-core manner.*

5.1 Architecture and WOFS Properties

Data layout. As Figure 5 shows, WOLVES manages PM at a 4 KiB block granularity and reserves the first two blocks for *superblock* and its redundant copy. The subsequent few blocks are reserved for bitmaps that persist *pkg-group* addresses for fast recovery (§4.5). The rest of the space stores data blocks and packages. WOLVES allows only one type of package in a *pkg-group* to reduce bitmap size and simplify space management. Thus, WOLVES reserves $\frac{4}{(4096\times8)} \approx 0.012\%$ PM space for bitmaps, where "4" denotes four types of atomic packages.

Space management. WOLVES adopts a two-level allocator (*tl-allocator*) for space allocation on each CPU, and each allocator manages a contiguous area R where $R = \frac{PM\ Size}{NumOfCPU}$. Specifically, the allocator maintains a list of free data blocks in the current area using a red-black tree. If no blocks are available, WOLVES allocates space by consulting other core allocators in a round-robin manner. To reclaim a data block, since the allocated space never crosses allocator boundaries, WOLVES determines the corresponding allocator by simple address division, and reinserts the block number into it.

To allocate packages, *tl-allocator* first allocates a data block as the *pkg-group* and then allocates a package in the group. Specifically, *tl-allocator* tracks each allocated *pkg-group* with a memory node, which uses a small 64-bit bitmap to manage its space (*i.e.,* 64=4096/64). The memory nodes are maintained in a mapping where the key is the address of the *pkg-group*, and the value is the node itself. To reclaim a package, *tl-allocator* first calculates the address of the *pkg-group*, then finds the tracked node according to the mapping, and clears the associated bits for further allocation.

PTL implementation and space reclamation. PTL builds the inode table as a global hash table with fine-grained per-

bucket locks for concurrency. It manages the data list and dent list in a per-CPU manner similar to *tl-allocator*, where each core maps inode numbers to the corresponding list for fast access. To manage *pkg-nodes* (in these lists) across different cores, high-level file system structures (*e.g.,* file/directory) index *pkg-nodes* to produce a global file system view. These structures are protected by per-file locks (*i.e.,* inode locks), ensuring concurrent consistency. The effects of written packages remain invisible until these high-level structures are atomically updated to reference the *pkg-nodes* via locking. Finally, PTL reclaims space by reasoning about operations along the critical path (§4.4). Note that the reclamation overheads are negligible as no extra PM I/O is required: PTL simply notifies the *tl-allocator* that the space is available for reuse.

Concurrency model. WOLVES follows the causal-ordering concurrency protocol [50] to align with WOFS's crash safety semantics, meaning that the independent packages can be persisted in parallel even if they are ordered by timestamp (*e.g.,* create two files under two different directories). For causally related packages, such as deleting a file after creating it, WOLVES serializes them for correctness. This is consistent with the virtual file system (VFS) framework.

5.2 Handling File System Operations

Directory abstraction and operations. Overall, the file system name hierarchy is based on VFS `dentry`-formed tree, so that WOLVES only maintains the parent-child name hierarchy: It deploys a 128-slot hash table for the directory inode to reorganize *dent list* based on name hashes. The table size is configurable to meet different demands. Given this organization, the operations for creating an inode, such as `mkdir`, can be achieved by writing a *create pkg* once using a `PCOMMIT`, generating a *C-node* in PTL, and updating the parent's hash table. Deleting an inode can be similarly achieved by writing an *unlink pkg* and removing related memory objects. For inode deletion/creation, *order map* should be updated or checked to track and reclaim the invalidated *unlink pkg*; moreover, the *create pkg* should be reclaimed after inode deletion (§4.4).

File abstraction and operations. WOLVES adopts dynamic arrays [16, 51] for files to reorganize *data list*. Therefore, the `read` operation is simple: WOLVES calculates the array slot for the given offset and copies the data to the user buffer. We now introduce how WOLVES handles the `write` operations.

- *Non-overlap write*. Typically, a non-overlap write is done by writing data using non-temporal stores (*i.e., movnti*) to bypass CPU cache [11], writing a *write pkg* once, inserting a new *W-node*, and updating the dynamic array.
- *Overlap write*. WOLVES writes data in a COW manner [21]. After generating a new *write pkg* $[c,d]$ and a *W-node* $[c,d]$, the overlapped *write pkg* (pointed by *W-node* ranging $[a,b]$) is handled in three ways: (1) if $[c,d] \supset [a,b]$, WOLVES reclaims the old *write pkg* and data blocks; (2) if $c < a, d < b$ (or $c > a, d > b$), WOLVES reclaims the overlapped data blocks and modifies the old *W-node* to $[d,b]$ (or $[a,c]$); (3) if $[c,d] \subset [a,b]$, WOLVES modifies the old *W-node* to $[a,c]$, and generates a new *write pkg* and *W-node* for $[d,b]$.
- *Append write*. For an append-like operation (*i.e.,* `offset` = `size`), instead of creating a *write pkg*, WOLVES appends the new data to the blocks that are already allocated to the file, and writes the changed file size in an 8-byte atomic reserved field of the *write pkg* header without recalculating the CRC32. To verify the validity of a *write pkg*, WOLVES first assigns zero to the reserved field and then checks CRC32.

Ops	Package (§4.2)	Implementation Details
`create`, `mkdir`, `mknod`	*Create pkg*	① Persist *create pkg*, ② update *inode map* and parent's hash table, ③ check *order map* to determine whether to reclaim an *unlink pkg* (§4.4).
`read`	*None*	① Search the file index to obtain data addresses, ② copy the data from PM to the user buffer.
`write`, `fallocate`	*Write pkg*	① Persist *write pkg*, ② update *data list* and file index, ③ reclaim overwritten *write pkg* (§4.4)
`unlink`, `rmdir`, `rm`	*Unlink pkg*	① Persist *unlink pkg*, ② unlink *C-node* from the hash table and *dentry list* and free *A-node*, ③ update *inode map* and *order map* (§4.4).
`link`	*Create pkg*	Similar to `create`, but changes the linked inode of the name entry to the linked one (§4.2).
`chmod`, `chown`, `truncate`	*Attr pkg*	① Persist *attr pkg*, ② replace the newly generated *A-node* for the inode, ③ reclaim the old *attr pkg* associated with the file (§4.4).
`rename`	*Create* + *Unlink*	Reuse `link` and `unlink` operations with preallocated *create* + *unlink pkg*. Assign a forward-pointer from *create pkg* to *unlink pkg*.
`symlink`	*Create* + *Write*	Reuse `create` and `write` operations with preallocated *create* + *write pkg*. Assign a forward-pointer from *create pkg* to *write pkg*.

Table 3: Implementation of 15 typical file system operations. *This table shows the four atomic package abstractions are fairly enough for most of the file operations.*

Other operations. The implementations of 15 typical file system operations with packages are shown in Table 3. Other operations, such as `cp`, are omitted as they are handled by VFS once WOLVES has implemented the above operations. Note that the implementation of these file operations may vary. For example, if a file system needs to maintain timestamps for `read`, it can simply write an *attr pkg* to update the timestamps.

5.3 Recovery

Atomic package recovery. For atomic packages, WOLVES reasons their causal ordering to fix inconsistencies and rebuild PTL: It first scans the *create* bitmap to locate *create pkg* to rebuild the inode table and then the *unlink* bitmap to reclaim the invalidated inode. *Dent list* is rebuilt along with the inode table by parsing the name entry in the *create pkg*. Next, WOLVES scans the *write* bitmap to recover *data list*s. When constructing *data list*s, the overlaps of *write pkg*s are reasoned based on their timestamp, and *W-node*s are updated accordingly. Finally, WOLVES scans the *attr* bitmap to update the attributes of the inode and reclaims the truncated data.

Compound package recovery. For the compound packages, WOLVES verifies integrity by checking the forward-pointers:

the compound package is valid unless all the linked packages are valid. This verification is integrated into the package scanning process. As an example, we consider the `rename` operation, which combines *create* and *unlink pkgs*. For brevity, we denote the *create* and *unlink pkgs* as $P_{c|r}$ and $P_{u|r}$, respectively. The recovery of `rename` proceeds in two phases.

First, *validate the first package during the create bitmap scan.* If a $P_{c|r}$ is found, follows its forward-pointer. If it points to a valid $P_{u|r}$, the rename operation is successful, and a mapping from $P_{u|r}$ to $P_{c|r}$ is inserted into a temporary hash table. In this case, $P_{c|r}$ is treated like a normal *create pkg*. Otherwise, the pointed package is invalid (*e.g.,* broken checksum) or is not a $P_{u|r}$, the pointer is deemed stale, and the $P_{c|r}$ is discarded. This is safe since a compound package can be reclaimed only if all its components are reclaimed. Second, *validate the second package during the unlink bitmap scan.* If a $P_{u|r}$ is found but no $P_{c|r}$ points to it (by checking the hash table), it is reclaimed. Otherwise, it is processed normally.

A similar process applies to other compound packages, though it is integrated into different package scanning phases. For example, the `symlink` operation requires first verifying the *create pkg* (*i.e., create pkg* scanning), followed by the *write pkg* (*i.e., write pkg* scanning), to ensure integrity.

Allocator recovery. During package scanning, WOLVES recovers the allocators by marking all space as free, then restoring the allocated space for valid packages and data blocks.

5.4 Other Optimizations

Huge allocation (HA). WOLVES allocates 2 MiB huge blocks for append-like writes. Hence, smaller writes (*e.g.,* KV) can be appended to the huge block by updating existing *write pkg* (see §5.2), which reduces PM buffer pollution by increasing buffer hits. If huge blocks are not available, WOLVES allocates as many contiguous blocks as possible.

Read ahead (RA). For `read`, WOLVES can prefetch the to-be-read data in an XPLine (*i.e.,* 256 bytes) stride for acceleration. Specifically, to read `size` data at `offset`, WOLVES first issues a number of $size/256$ `prefetcht0` from `offset` with a 256-byte stride; then, it copies the data to the user buffer. This enables copy I/O to be overlapped with the prefetching I/O.

6 Evaluation

We design experiments on WOLVES (*i.e.,* our WOFS implementation) to answer the following questions:

- Can WOLVES achieve robust crash consistency?
- Can WOLVES fully release the I/O performance of the PM and achieve optimal metadata operation performance?
- How does WOLVES perform under the aged scenarios?
- How expensive are WOLVES recovery procedures, memory overheads, and storage space consumption?
- How does WOLVES compare to (synchronous) soft update?
- Does the WOFS/WOLVES design generalize to future platforms beyond Intel PM (*e.g.,* Memory-Semantic SSD [3])?

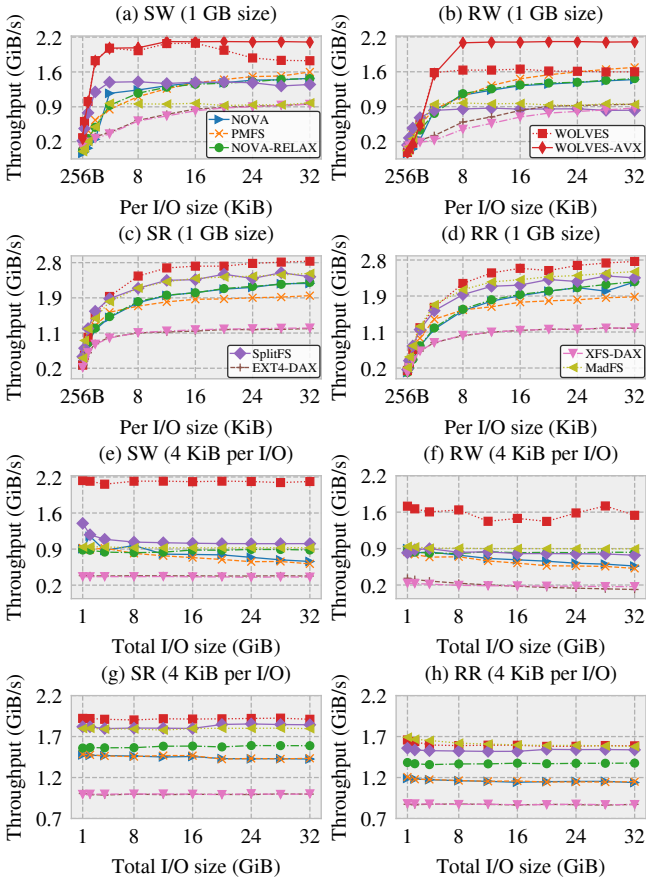


Figure 6: I/O performance comparison under different I/O patterns. *Sub-figures (a)–(d) study the I/O throughput varying per I/O size when populating a 1 GiB file. Sub-figures (e)–(h) study the I/O scalability under different file sizes with per I/O size 4KiB.*

6.1 Experimental Setup

Testbed. The experiments are conducted on a server with a 16-core Intel Xeon Gold 5218 CPU, 128 GiB DRAM, and 2×256 GiB Intel Optane PM configured in non-interleaved mode[4], running atop Linux Kernel 5.1.0 [52].

Competitors. We compare seven PM file systems that apply different synchronous crash consistency techniques: PMFS, NOVA, NOVA-RELAX (NOVA w/o data atomicity), SplitFS, MadFS, EXT4-DAX, and XFS-DAX. PMFS is equipped with a huge-page-aware technique from WineFS [18] to resist file system aging. SplitFS is configured in POSIX mode for the highest performance. MadFS [28] is a recent user-space PM LFS built atop EXT4-DAX. The last two file systems are production-ready and mounted in `ordered` mode. We also compare WOLVES with soft update file systems in §6.11, including (asynchronous) SoupFS [53], (asynchronous) HUNTER [51], and (synchronous) SquirrelFS [25]. Note that despite our best efforts, we only successfully run MadFS in FIO, single-threaded Filebench, and several FxMark work-

[4]Note that the interleaved mode shows similar performance trends. We thus only report non-interleaved mode due to page limits.

%ile (us/op)	WOLVES	NOVA	PMFS	SplitFS	MadFS
90%	2.21	5.22	5.00	1.76	1.70
99%	3.16	7.12	6.00	4.24	4.08
99.9%	12.84	19.38	21.00	951.09	1095.00
99.99%	18.92	23.27	329.4	1125.58	1302.53

Table 4: Tail latency comparison. *The table is measured under the single-threaded 32 GiB SW workload with 4 KiB per I/O.*

loads, using their provided configurations.

Methodology. We ran each test at least five times and reported the *average value*. The standard deviation was less than 5%, suggesting the experiments are repeatable.

6.2 Crash Consistency

We have used a formal logical model [37, 54, 55] to prove that WOLVES can provide robust crash consistency. The proof is attached to the Appendix A. In this subsection, we measure if WOLVES recovers correctly against crashes via three steps: (1) run workloads for specific I/O patterns, (2) trace I/Os and reorder them, and (3) create images with random crash points and remount WOLVES. We manually trace PM I/Os at the instruction level, including *store+clwb*, *movnti*, and *sfence*; traces are written to a file for reordering (between *sfences*) and replaying (until a random crash point).

Exhaustive crash consistency verification is practically infeasible due to the infinite space of workloads. Thus, we adopt three representative workloads, following OptFS [15] and BarrierFS [13], to show that WOLVES can recover from common random crash scenarios: an append workload that keeps appending 4 KiB data to a file, `create_delete` that creates then deletes 10 files, and `rename_root_to_sub` that creates and renames (mv) a file to a sub-dir. These workloads cover most I/O patterns in WOLVES: $D \rightarrow P|(m_{cp})$, $P|(m_{cp})$, and $P_{unlink}|P_{create}$. To see if WOLVES recovers to the latest consistent state, we create checkpoint for every file operation and compare the recovered system with the latest checkpoint. For each workload, we test 1000 random crash points. The results show that WOLVES consistently recovers to the latest consistent state prior to the crash, suggesting that it can correctly handle most common crash scenarios.

6.3 I/O Performance

We leverage FIO [56] to study WOLVES I/O characteristics.

I/O patterns. As Figures 6a–6d show, all file systems ramp up throughput with the per I/O size increases until the PM buffer is exhausted [10, 57]. For SW/RW, WOLVES shows significant benefits when the per I/O size is less than 16 KiB since it writes package once, but collapses afterward due to the deficiency of `movnti`, making data I/O dominated [23]. We address this by integrating *vmovntdq* into kernel (*i.e.,* WOLVES-AVX), and the throughput scales to PM bandwidth limits with the increasing I/O size. In SR/RR, WOLVES demonstrates the read performance of a kernel file system can even exceed that of user-space SplitFS thanks to the read-ahead (RA) technique. Notably, without RA (not shown in the figure), WOLVES has a similar `read` performance to other PM kernel file systems due to their similar `read` code paths.

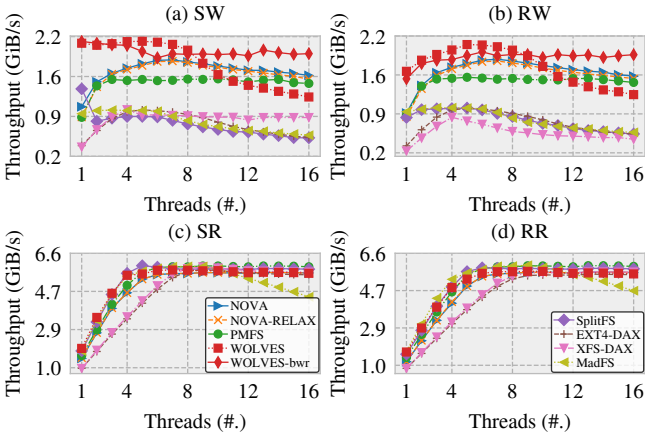


Figure 7: Concurrency performance comparison. *The experiment compares the 4 KiB per I/O write throughput of different PM file systems under 1–16 threads. Each thread populates a distinct 1 GiB file under the same directory.*

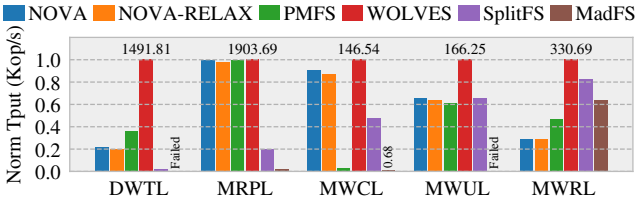


Figure 8: FxMark evaluation. *DWTL reduces the file size by 4 KiB; MRPL opens a file in five-depth dirs; MWCL creates an empty file; MWUL unlinks an empty file, and MWRL renames a file.*

File size scalability. As Figures 6e and 6f show, in the SW workload, the write throughput of WOLVES stabilizes at 2.20–2.24 GiB/s, achieving 97.3%–99.1% of the throughput of raw PM [2]. WOLVES fails to achieve similar performance in RW as RW cannot benefit from the huge allocation (HA), but it still achieves 1.65–9.44× throughput compared to others due to the minimized metadata I/Os and ordering points for the metadata write-once scheme.

Tail latency. As Table 4 shows, WOLVES achieves the lowest tail latency compared to NOVA and PMFS, but SplitFS and MadFS outperform WOLVES before the 99.0 percentile since they persist data in the user space. However, SplitFS and MadFS suffer severe tail latency afterward due to their occasional kernel metadata access, which becomes bottlenecks.

Concurrency. As shown in Figure 7, WOLVES outperforms competitors in 1–8 threads. Nevertheless, WOLVES suffers performance degradation when thread numbers ≥ 9. The observation is consistent with prior reports that PM has limited concurrency due to its hardware contentions [11, 16, 17, 51, 57]. To address this, we sample the I/O latency and insert delay when the I/O latency > 3000 ns to alleviate the contention

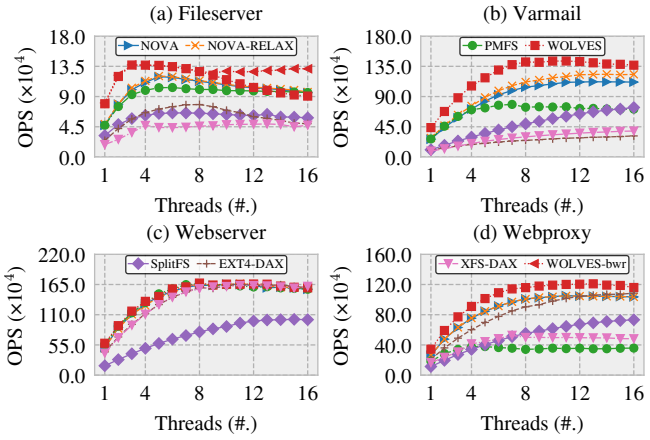


Figure 9: Filebench evaluation. *Each workload runs for 60 seconds under 1–16 threads. Operations per second (OPS) is reported.*

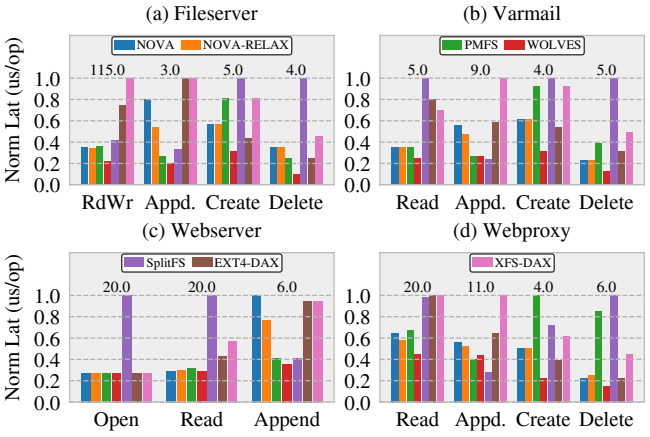


Figure 10: Filebench operation latency breakdown. *We report single-threaded performance breakdown. The numbers above the bars show WOLVES latency. Note that the open latency is similar under four workloads, and thus we only report it under Webserver.*

(*i.e.,* WOLVES-bwr). Consequently, WOLVES can scale to PM bandwidth limits with the increasing thread numbers.

6.4 Metadata Operation Performance

We study metadata operations using FxMark [58] under a single thread. For the concurrent metadata operations, VFS' locking scheme limits the parallelism [16,35], and optimizing its locking is beyond the scope of this paper. Figure 8 shows that WOLVES achieves the highest throughput for most metadata operations thanks to the metadata (*i.e.,* package) write-once scheme. In the MRPL workload, all the evaluated file systems, except for SplitFS and MadFS, show similar throughput since they rely on VFS directory traversing. SplitFS and MadFS, however, suffer from their own traversing and in-kernel traversing, thereby reducing the overall throughput.

6.5 Macrobenchmarks

We evaluate four workloads in Filebench [59] (see Table 2). As Figure 9 shows, WOLVES achieves the highest throughput under both write- and read-intensive workloads. In Fileserver, WOLVES can alleviate I/O contention by regulating bandwidth (*i.e.,* WOLVES-bwr). Figure 10 studies the latency breakdown of critical file operations. WOLVES achieves the lowest latency when performing file creation/deletion due to its metadata write-once scheme. We observe that PMFS has a similar append performance as WOLVES because PMFS uses a similar append approach. SplitFS, however, suffers severe metadata overheads due to its in-kernel metadata operations.

We also compare WOLVES with MadFS in Table 5 and find that MadFS suffers extremely low throughput due to its heavy metadata overheads, which is consistent with FxMark results.

Workload	MadFS	WOLVES	Speedup
Fileserver	5.57 Kops/s	80.4 Kops/s	14.4×
Varmail	7.13 Kops/s	438 Kops/s	61.4×
Webserver	63.6 Kops/s	581 Kops/s	9.14×
Webproxy	9.6 Kops/s	344 Kops/s	35.8×

Table 5: Filebench comparison with MadFS. *Here, we report single-threaded Filebench performance for MadFS.*

6.6 Performance Breakdown

I/O breakdown. Comparing Figure 11 with Figure 2, WOLVES reduces latency by 5%–86% compared to NOVA, SplitFS, and PMFS. Furthermore, *CP* overhead is less than 0.5%. Notably, in SW, WOLVES reduces data time by 42.2%; the reason is that the metadata write-once and HA reduce XPBuffer pollution; as a result, even the metadata time accounts for nearly 40%, it can achieve more than 97% of PM bandwidth. Using *ipmctl*, we confirm that WOLVES reduces per-operation metadata I/O by 70%–17.3× compared to PMFS and NOVA.

Technique breakdown. Figure 12 studies how the metadata write-once (WO), huge allocation (HA), and read ahead (RA) contribute to WOLVES. First, the metadata write-once scheme improves 1.01–1.66× and 72%–1.30× throughput under SW and RW compared to other PM file systems, which is the main source of performance improvement. We observe that sometimes WOLVES can reach 2.01 GiB/s under SW, primarily because the CPU has the potential to reorder *movnti*s (for data I/O, §5.2) to improve I/O parallelisms and reduce PM contention. Second, HA further improves throughput by 20.2% for SW, but it is less effective for RW due to the lack of sequential I/O patterns in RW. Finally, RA can improve SR and RR by 22%–27% due to the improved read parallelism.

6.7 Real-world Applications

LevelDB. Figure 13a shows the performance of different PM file systems on LevelDB using YCSB workloads (driven by built-in `db_bench`) [20]. WOLVES achieves the most significant improvement under write-intensive workloads (Load A, E, and Run A). On the read-intensive workloads, WOLVES's throughput is still higher than others, thanks to the RA.

RocksDB. Figure 13b runs RocksDB [23] (1M key-value pairs), which is configured with no preallocate. Despite our

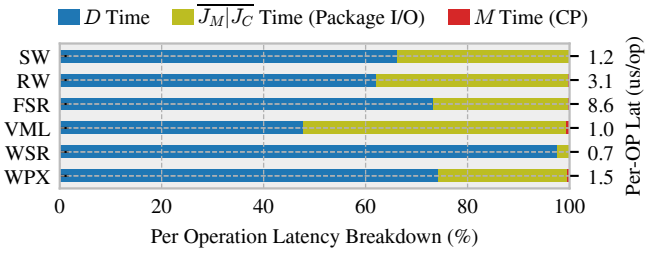


Figure 11: WOLVES I/O breakdown. *SW and RW are both 32 GiB with 4 KiB per I/O. FSR, VML, WSR, and WPX are short for fileserver, varmail, webserver, and webproxy running for 60 seconds.*

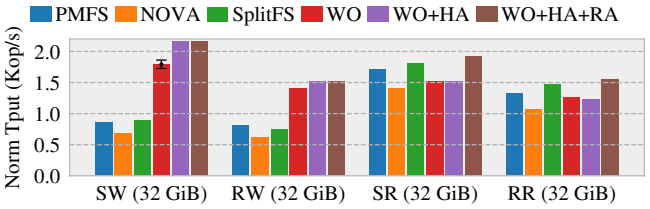


Figure 12: WOLVES techniques breakdown. *We show the single-threaded I/O performance with 4 KiB per I/O. WO denotes WOLVES without HA and RA. WO+HA+RA is the final WOLVES.*

best efforts, we fail to run SplitFS [20] on RocksDB. WOLVES has 1.26–6.73×, 1.36–5.21×, 1.26–3.93×, and 1.20–4.46× the throughput of other PM file systems for the Sequential Fill, Random Fill, Random Append, and Random Update.

6.8 Aging and Fragmentation

During frequent file creations/deletions (*i.e.,* aging), WOLVES incurs data/package fragmentation. To retrieve WOLVES aging performance, we use Geriatrix and widely cited Agrawal profile [60] to age WOLVES at 80% PM utilization until convergence in relative age distribution [61]. Following aging process, we stress SW and RW on WOLVES.

Figure 14a suggests that under the SW, (1) WOLVES has an unstable throughput 1.85–1.95 GiB/s due to fragmentation; (2) the performance degradation is minor as ∼50% of the Agrawal profile's files are ≥8 KiB, and their deletions free up many contiguous space (*e.g.,* more than 50% free contiguous block numbers are in $[19,384]$), which is then leveraged by HA to alleviate fragmented package writes. To show an extreme case, we set all files in the Agrawal profile to 4 KiB, resulting in an extremely fragmented space (*e.g.,* all available contiguous block numbers ≤8). However, even under this profile, WOLVES still outperforms NOVA and PMFS, and obtains 1.70–1.82 GiB/s throughput thanks to its metadata write-once scheme. Figure 14b shows that under the RW, WOLVES throughput is 1.31–1.44 GiB/s. Here, the performance degradation mainly comes from the fragmented packages as WOLVES has no HA optimization for RW.

In summary, though performance degradation occurs during aging, WOLVES still outperforms other PM file systems. Note that WOLVES does not apply any defragmentation techniques currently, which we plan to explore as a future work.

Category	Workload	NOVA	DR	DR-OPT	WOLVES
Large file	FIO-32G	24.2 s	70.1 s	60.4 s	2.61 s
Write intensive	Fileserver	2.48 s	71.9 s	45.8 s	3.99 s
Read intensive	Webserver	2.52 s	70.5 s	43.1 s	2.75 s

Table 6: Failure recovery time. *DR dumps package addresses into bitmaps when unmount; DR-OPT tags unallocated blocks during format and skips scanning them when recovery. Note that (write-intensive) Varmail and (read-intensive) Webproxy have similar results to Fileserver and Webserver, thus we omit them in the table.*

6.9 Recovery Overhead

Common scenarios. Table 6 compares the failure recovery time of NOVA, DR, DR-OPT, and WOLVES under three representative evaluated workloads. Thanks to *CP* (§4.5), WOLVES can only scan bitmaps and *pkg-group*s, thus achieving a failure recovery time comparable to that of NOVA. Note that the extremely slow recovery of NOVA under FIG-32G is due to its engineering deficiency: For files larger than 1 GiB, NOVA must scan its inode-log multiple times, as it can only replay entries for 1 GiB of data per pass.

Worst scenario. We build a worst-case workload to show that WOLVES can recover quickly thanks to the metadata-only scanning: We assume the file system is fully populated with non-empty files (*i.e.,* PM is fully occupied). Each file is written with a single data block, modified via `chown`, and eventually deleted. This process generates for each PM data block one 256-byte *create pkg*, one 64-byte *write pkg*, one 64-byte *attr pkg*, and one 64-byte *unlink pkg*, leading to the highest metadata (packages) scanning overhead. Even in this case, WOLVES scans only around (256+64+64+64)/4096=10.9% of the total 256 GiB PM space with ∼60 million files (which is similar to NOVA), and the overall recovery time for WOLVES is ∼21.6 seconds. We believe that this overhead is acceptable as it remains faster than the traditional fsck for SSD file systems, which can take more than 2 miniutes for 60 million files [62–64], thanks to PM's high read bandwidth (∼2 GiB/s).

6.10 Resource Consumption

Memory overhead. We measure memory usage by hooking in-kernel memory allocation. In Fileserver and Varmail, WOLVES and NOVA achieve comparable memory consumption: WOLVES uses ∼8.93 MiB and ∼10.82 MiB memory, while NOVA uses 10.48 MiB and 8.31 MiB memory. Unlike NOVA, which consumes no memory for closed files, WOLVES maintains 3.8 MiB–5.12 MiB of memory for PTL even when all files are closed. This modest memory overhead (*i.e.,* around 0.3%–1.6% of the workload's size) enables significant performance benefits, with WOLVES achieving 68% and 65% higher throughput in Fileserver and Varmail, respectively.

Space overhead. Adding headers for packages incurs extra space overheads, but we argue that it is acceptable: conducting a total of 128 GiB 4 KiB I/Os requires (1K+1)–(512K+1) *pkg-group*s for one *create pkg* plus 64K–32M *write pkg* (with or w/o HA), leading to 0.0015%–0.7% space in a 256 GiB PM.

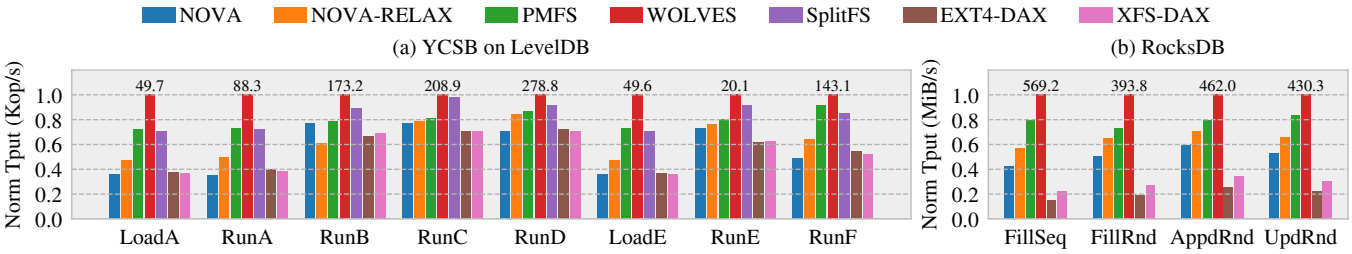


Figure 13: Real-world application performance comparison. *We measure the performance using the application's built-in db_bench tool. The numbers above the bars indicate the absolute throughput of WOLVES.*

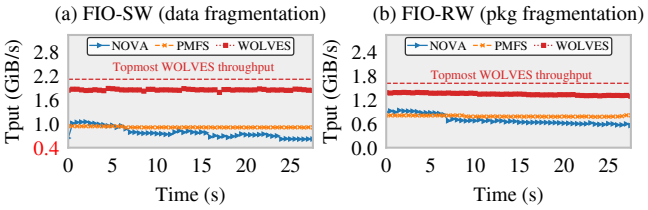


Figure 14: Aging/Fragmentation performance. *We measure the single-threaded sequential/random write throughput with 4 KiB per I/O under the Agrawal-profile-aged file system.*

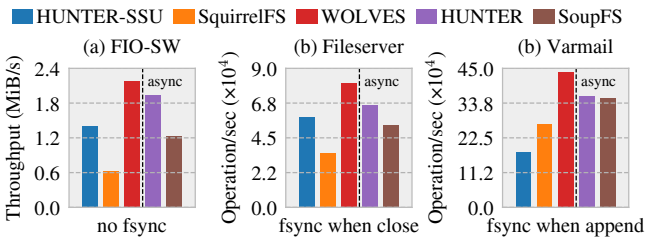


Figure 15: Comparison with (synchronous) soft update. *We hack HUNTER to support ordered synchronous soft update (i.e., HUNTER-SSU). SquirrelFS is written in Rust to support compile-time consistency check, and is measured under Linux kernel 6.3.0.*

6.11 Case Study: Beyond (Sync.) Soft Update

We use workloads with no/heavy/real-`fsync` to show that WOFS crash consistency outperforms SSU and even asynchronous soft update (ASU): (1) Figure 15a shows that ASU can outperform most of SSU, but WOLVES outperforms ASU as the former minimizes metadata I/Os while the latter periodically flushes metadata orderly, interfering with foreground I/Os. (2) By modifying FIO to issue `fsync` for every system call, we emulate the I/O behavior in RocksDB. The results (not shown) suggest that SSU has no performance degradation, while HUNTER drops to 762 MiB/s due to synchronization and I/O overheads. (3) Figures 15b and 15c show that WOLVES outperforms ASU by 21%–52% in workloads with occasional `fsync`, while other SSU suffers severe overheads due to many small, random, and ordered metadata I/Os.

6.12 Case Study: Memory-Semantic SSD

In this subsection, we show the generality of WOFS/WOLVES on Samsung Memory Semantic (MS) SSD [3], which is an emerging commercial PM backed by SSD. To simulate MS-SSD, we use *io_uring* [65] (with 128 depths) to emulate the byte-addressable interface, which allows the host to issue I/O to the *io_uring* buffers using loads/stores. The buffer is protected by a Castle UPS [66] and regarded as *persistent buffers*. The backend is a 1.6 TB Intel DC P4610 Series SSD; the issued I/Os are digested to SSD in a 4 KiB I/O unit through background threads (using `O_DIRECT`). Under this platform, the maximal write throughput is around 1900 MiB/s.

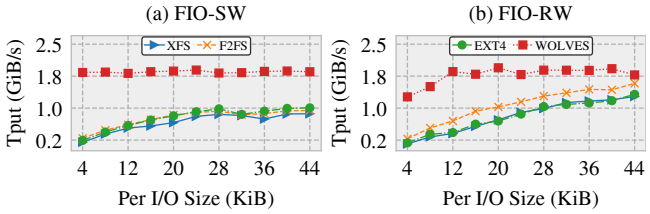


Figure 16: Performance comparison in MS-SSD. *We measure sequential/random write throughput in a single thread. The maximal write throughput of MS-SSD is ∼1900 MiB/s.*

We add several lines of code to emulate a PM-like interface, and thus porting WOLVES to the MS-SSD platform is trivial. We compare the write throughput of WOLVES with XFS, EXT4 and F2FS using FIO. For fairness, they are all configured to use direct-I/O and *io_uring* engine without `fsync` calls. Unsurprisingly, Figure 16 verifies similar results as in the Intel PM platform that WOLVES can achieve the highest write throughput thanks to the metadata write-once scheme.

7 Discussions and Future Work

Why do prior file systems manage metadata separately? Separating file system metadata with different responsibilities, such as inode, bitmap, and dentry, inherits the old wisdom of file system designs [67, 68]. Modern LFS, F2FS [45], also employs multiple tables (*e.g., SIT* and *NAT*) to manage metadata objects. Such design enables simplified crash consistency, easy/clear file abstraction, and avoidance of the wandering tree in flash memories [45]. However, it meets limitations for exploiting fast synchronous PM I/Os.

Distinguishing from LFS. The key novelty of WOFS lies in generating the specific, aggregated package for each file oper-

ation, which is different from the scattered metadata produced by modern LFSes, such as NOVA [21] and F2FS [45]. This write-once scheme minimizes metadata I/Os, resulting in a considerable improvement over LFS competitors. Moreover, WOFS's non-log layout and immediate package reclamation also distinguish it from LFS. These designs facilitate PM's relatively good random I/O (compared to HDDs/SSDs) and avoid significant GC overhead typically associated with LFS.

Shall WOFS eliminate data ordering? It is charming to eliminate data ordering via data checksum [15] (*i.e.,* $\overline{D|J_M|J_C}$). However, we find that data checksums, which cause user-to-kernel memory copies and computation overheads on large content, outweigh the benefits of eliminating ordering in fast PM. Specifically, we find that using CRC32 and faster xxHash [57,69] as data checksum inversely adds ∼40.1% and ∼32.3% overheads in the write path. Moreover, WOFS can already reach the PM bandwidth limit without data checksum. Hence, we leave exploring data checksums for future work.

Shall WOFS use checkpoint to accelerate recovery? Periodically checkpointing the PTL can alleviate package scan/rebuild overhead. However, we do not intend to use it for three reasons. First, checkpointing interferes with foreground PM I/Os, thereby degrading file system throughput [70], which we aim to avoid. Second, package/metadata occupy a small portion of storage, and scanning them is fast as no data block involved [21]. Finally, PM enables a much faster scanning performance than SSDs/HDDs, and reconstructing PTL is also fast as it merely manipulates DRAM. As a result, WOFS can recover in less than 30 seconds even in the worst scenario (§6.9). We do acknowledge the opportunity to checkpoint the PTL during system idle time, which we leave for future work.

Can memory overheads in WOFS be reduced? If memory overhead for PTL maintenance is a concern (*e.g.,* for memory-sensitive scenarios), it is feasible to use a PM-DRAM hybrid approach for PTL to reduce memory overheads while retaining performance, since PTL does not require flush/fence for ordering and durability. Specifically, we allocate 25%, 50%, 75%, and 100% of PTL metadata directly from PM and find that the performance degradation is negligible in SW, while only 3.1%, 6.3%, 8.8%, and 12.5% degradation in RW.

Performance considerations for compound packages. Currently, even the most complex operation [21], such as `rename`, involves only two packages, and the overhead of writing them concurrently with the forward-pointer is comparable to that of writing a single package. Figure 8 also shows that WOLVES's `rename` operation significantly outperforms existing PM file systems. Besides, such operations are infrequent. Particularly, `rename` operations account for ∼1%–10% of operations in many realistic/synthetic workloads [71,72].

8 Related Work

Optimizing ordering points. NOFS leverages back-pointers to eliminate the order between metadata and data [37]. OptFS introduces data checksum to eliminate the ordering between data and transactions for slow HDD [15]. BarrierFS redesigns the block I/O stack to enforce I/O order through multi-layer collaboration without waiting for persistence [13]. *Hardware Transactional Memory (HTM)* is an attractive hardware technique to eliminate software ordering. However, HTM-based crash consistency is currently only applicable on the eADR platform [24], which limits its usage. In contrast, our WOFS redesigns metadata as packages to minimize metadata I/Os and ordering points for fast synchronous crash consistency.

Metadata-optimized file systems. BPFS [73] novelly employs shadow paging that uses a COW scheme for metadata updates, but it leads to cascade of updates, which can be even more costly than JFS [21]. TableFS [74] and BetrFS [75] deploy write-optimized indexes to defer and batch small metadata writes [76]. Several works aim to use supercapacitors to protect asynchronous metadata I/Os, but these devices are not widely available [27,77]. In contrast, WOFS generates specific metadata for a file operation as one compact package to minimize metadata overheads without trading off durability.

Data-optimized file systems. OdinFS [17] and ArckFS [16] are two PM file systems designed to stripe data across multiple nodes to avoid single PM contention. However, these works follow traditional crash consistency mechanisms, leading to unorchestrated metadata I/Os. As a result, ArckFS (which even updates metadata in the user space) achieves only around 1400 MiB/s throughput under the single-threaded SW in our machine. We believe that these works are complementary to WOFS for machines with multiple nodes and PM arrays.

9 Conclusion

We introduce WOFS, a new file system model for fast and synchronous crash consistency. WOFS generates and manages specific metadata for a file operation as a checksum-protected package, and writes it once with a single ordering point. Experiments driven by various benchmarks and applications show that WOLVES, a WOFS prototype, outperforms existing file systems on PM platforms (*e.g.,* both Intel PM and MS-SSD), and can potentially reach PM bandwidth limits. The source code and test scripts are publicly available at `https://github.com/WOFS-for-PM/`.

Acknowledgment

We sincerely thank our shepherd, Jon Howell, and the anonymous reviewers in OSDI for their constructive comments and insightful suggestions. This research was partly supported by the National Key R&D Program of China under Grant 2023YFB4502900, National Natural Science Foundation of China under Grant 62472127, Shenzhen Science and Technology Program under Grants RCYX20210609104510007, GXWD20231128111309001 and KJZD20230923114610021, and GuangDong Basic and Applied Basic Research Foundation under Grant 2023A1515110072.

References

[1] Intel Announces Optane Storage Brand for 3D XPoint Products. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products, 2015.

[2] Intel@ Optane DC^{TM} Persistent Memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, 2019.

[3] Samsung Electronics Unveils Far-Reaching, Next-Generation Memory Solutions at Flash Memory Summit 2022. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022, 2022.

[4] Understand How the CXL SSD can Aid Performance. https://www.techtarget.com/searchstorage/feature/Understand-how-the-CXL-SSD-can-aid-performance, 2023.

[5] Jonas Rabenstein et al. On the Performance of NVRAM-based Operating Systems: A Case Study with Linux and FreeBSD. https://doi.org/10.25593/issn.2191-5008/CS-2023-01, 2023.

[6] Available First on Google Cloud: Intel Optane DC Persistent Memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory., 2018.

[7] For Enterprise Storage, Persistent Memory is Here to Stay. https://www.networkworld.com/article/3398988/for-enterprise-storage-persistent-memory-is-here-to-stay.html, 2019.

[8] Tao Cai, Yueming Ma, Peiyao Liu, Dejiao Niu, and Lei Li. A New NVM Device Driver for IoT Time Series Database. *Micromachines*, 13(3):385, 2022.

[9] Persistent Memory-based Storage Node for HPC Domain. https://www.snia.org/educational-library/persistent-memory-based-storage-node-hpc-domain-2022, 2022.

[10] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its On-DIMM Buffering. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, pages 488–505, Rennes, France, 2022.

[11] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, Santa Clara, CA, USA, 2020.

[12] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *The VLDB Endowment (PVLDB)*, 14(4):626–639, 2021.

[13] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO stack for flash storage. In *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 211–226, Oakland, CA, USA, 2018.

[14] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The Next Generation of Ext2/3 Filesystem. In *Proceedings of 2007 Linux Storage & Filesystem Workshop (LSF)*, pages 1–36, San Jose, CA, USA, 2007.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 228–243, Farmington, PA, USA, 2013.

[16] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 150–165, Koblenz, Germany, 2023.

[17] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193, Carlsbad, CA, USA, 2022.

[18] Rohan Kadedodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Greg Ganger, Aasheesh Kolli, and Vijay Chidambaram. WineFS: a Hugepage-aware File System for Persistent Memory that Ages Gracefully. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual Event, Germany, 2021.

[19] Edwin H.-M. Sha, Xianzhang Chen, Qingfeng Zhuge, Liang Shi, and Weiwen Jiang. A New Design of In-Memory File System Based on File Virtual Address Framework. *IEEE Transactions on Computers (TC)*, 65(10):2959–2972, 2016.

[20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 494–508, Huntsville, Ontario, Canada, 2019.

[21] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, Santa Clara, CA, USA, 2016.

[22] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, pages 1–15, Amsterdam, The Netherlands, 2014.

[23] R. Li, R. Xiang, Z. Xu, et al. ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, pages 35–50, Santa Clara, CA, USA, 2022.

[24] Jifei Yi et al. HTMFS: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–34, Santa Clara, CA, USA, 2022.

[25] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. SquirrelFS: Using the Rust Compiler to Check File-System Crash Consistency. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 387–404, Santa Clara, CA, USA, 2024.

[26] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 478–493, Huntsville, Ontario, Canada, 2019.

[27] Yanqi Pan, Hao Huang, Yifeng Zhang, Wen Xia, Xiangyu Zou, and Cai Deng. Delaying Crash Consistency for Building A High-Performance Persistent Memory File System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 43(9):2620–2634, 2024.

[28] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. MadFS:Per-File Virtualization for Userspace Persistent Memory Filesystems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*, 2023.

[29] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[30] Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso. FastCommit: Resource-efficient, Performant and Cost-effective File System Journaling. In *Proceedings of 2024 USENIX Annual Technical Conference (USENIX ATC)*, pages 157–171, Santa Clara, CA, 2024.

[31] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? In *Proceedings of 2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 753–767, Virtual Event, USA, 2020.

[32] Tomas Vondra. PostgreSQL vs. fsync. How is it possible that PostgreSQL used fsync incorrectly for 20 years, and what we'll do about it. https://archive.fosdem.org/2019/schedule/event/postgresql_fsync/, 2019.

[33] Filesystems and Crash Resistance. https://lwn.net/Articles/788938/, 2019.

[34] Marshall K McKusick, Gregory R Ganger, et al. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of 1999 USENIX Annual Technical Conference, FREENIX Track (USENIX ATC)*, pages 1–17, Monterey, CA, USA, 1999.

[35] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jiwu Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 81–95, Virtual Event, USA, 2021.

[36] Re: [PATCH v10 00/21] Support ext4 on NV-DIMMs. https://lwn.net/Articles/610184/, 2014.

[37] Vijay Chidambaram et al. Consistency Without Ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, San Jose, CA, USA, 2012.

[38] Richard Fackenthal, Makoto Kitagawa, Wataru Otsuka, Kirk Prall, Duane Mills, Keiichi Tsutsui, Jahanshir Javanifard, Kerry Tedrow, Tomohito Tsushima, Yoshiyuki Shibahara, and Glen Hush. 19.7 A 16Gb ReRAM with 200MB/s Write and 1GB/s Read in 27nm Technology.

In *Proceedings of 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 338–339, San Francisco, CA, USA, 2014.

[39] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. *ACM Special Interest Group on Computer Architecture (SIGARCH)*, 37(3):2–13, 2009.

[40] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *Proceedings of 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 111–124, Seoul, Korea (South), 2021.

[41] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *Proceedings of 2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 603–616, Renton, WA, 2019.

[42] Vijayan Prabhakaran, Lakshmi N Bairavasundaram, Nitin Agrawal, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 206–220, Brighton, United Kingdom, 2005.

[43] David Woodhouse. JFFS: The Journalling Flash File System. In *Proceedings of 2001 Ottawa Linux Symposium (OLS)*, pages 1–12, Ottawa, Canada, 2001.

[44] UBIFS: Unsorted Block Images File System. http://www.linux-mtd.infradead.org/doc/ubifs.html, 2006.

[45] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, Santa Clara, CA, USA, 2015.

[46] Ipmctl. https://github.com/intel/ipmctl, 2018.

[47] Libfuse. https://github.com/libfuse/libfuse, 2023.

[48] linux/fs.h. https://elixir.bootlin.com/linux/v6.10.7/source/include/linux/fs.h, 2024.

[49] Younghwan Go, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Reliable, Consistent, and Efficient Data Sync for Mobile Apps. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 359–372, Santa Clara, CA, USA, 2015.

[50] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing (DISC)*, 9(1):37–49, 1995.

[51] Yanqi Pan, Yifeng Zhang, Wen Xia, Xiangyu Zou, and Cai Deng. HUNTER: Releasing Persistent Memory Write Performance with A Novel PM-DRAM Collaboration Architecture. In *Proceedings of the 60th Annual Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, 2023.

[52] SplitFS-5.1. https://github.com/rohankadekodi/SplitFS-5.1.git, 2023.

[53] Mingkai Dong and Haibo Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In *Proceedins of 2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 719–731, Santa Clara, CA, USA, 2017.

[54] Muthian Sivathanu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Somesh Jha. A Logic of File Systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pages 1–15, San Francisco, CA, USA, 2005.

[55] Vijay Chidambaram et al. Consistency Without Ordering. *Technical Report 1709, University of Wisconsin-Madison Computer Sciences*, 2012.

[56] Flexible I/O Tester. https://github.com/axboe/fio, 2017.

[57] Jiansheng Qiu, Yanqi Pan, Wen Xia, Xiaojia Huang, Wenjun Wu, Xiangyu Zou, Shiyi Li, and Yu Hua. LightDedup: A Light-weight Inline Deduplication Framework for Non-Volatile Memory File Systems. In *Proceedings of 2023 USENIX Annual Technical Conference (USENIX ATC)*, pages 101–116, Boston, MA, USA, 2023.

[58] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 71–85, Denver, CO, USA, 2016.

[59] Filebench File System Benchmark. http://sourceforge.net/projects/filebench, 2016.

[60] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-system Benchmarking. *ACM Transactions on Storage (TOS)*, 5(4):1–30, dec 2009.

[61] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging What You See and

What You don't See. A File System Aging Approach for Modern Storage Systems. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 691–704, Boston, MA, July 2018.

[62] David Domingo and Sudarsun Kannan. pFSCK: Accelerating File System Checking and Repair for Modern Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 113–126, Santa Clara, CA, USA, 2021.

[63] Ao Ma, Charlotte Dragga, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–15, San Jose, CA, USA, 2013.

[64] Marshall Kirk McKusick. Improving the Performance of FSCK in FreeBSD. *login Usenix Mag.*, 38(2), 2013.

[65] The Rapid Growth of io_uring. https://lwn.net/Articles/810414/, 2020.

[66] Castle 3C Series UPS (10 20kVA). https://www.santak.com/product/online-ups-10~20k.html, 2024.

[67] Remy Card. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux (DISL)*, Amsterdam, The Netherlands, 1995.

[68] Dr. Marshall Kirk McKusick. Keynote Address: A Brief History of the BSD Fast Filesystem. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, 2015.

[69] xxHash: Extremely Fast Non-cryptographic Hash Algorithm. https://github.com/Cyan4973/xxHash.git, 2023.

[70] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscSort: External Sorting For Byte-Addressable Storage. *Proceedings of VLDB Endow (PVLDB)*, 16(9):2103–2116, 2023.

[71] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, pages 313–328, Santa Clara, CA, USA, 2022.

[72] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. ScaleXFS: Getting Scalability of XFS Back on the Ring. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, pages 329–344, Santa Clara, CA, USA, 2022.

[73] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, Montana, USA, 2009.

[74] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC)*, pages 145–156, San Jose, CA, USA, 2013.

[75] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: A Right-Optimized Write-Optimized file system. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, Santa Clara, CA, USA, 2015.

[76] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, et al. Optimizing Every Operation in a Write-Optimized File System. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Santa Clara, CA, USA, 2016.

[77] Dushyanth Narayanan and Orion Hodson. Whole-System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, England, UK, 2012.

A Proof of Crash Consistency in WOLVES

WOLVES achieves fast and synchronous crash consistency. In terms of crash consistency, there are three advanced levels: (1) metadata consistency that avoids dangling files, duplicate pointers, and storage leaks [37], (2) data consistency that further guarantees a file will not contain data blocks belonging to another file [54], and (3) version consistency that guarantees the version of metadata matches with the version of pointed data blocks [37]. This appendix focuses on formally proving that the proposed WOLVES based on the metadata write-once scheme can achieve both data consistency and version consistency[5] by using the existing logical framework [54, 55].

A.1 Notations

File system entities are *containers*, *pointers*, and *generations*. In WOLVES, package and data blocks are containers, which can be freed and reused. Containers are linked to each other by pointers (*e.g., write-pkg* maps data blocks). The *generation* of a container is incremented after each allocation while the

[5] Data consistency is a stricter form of metadata consistency [37].

epoch of a container is incremented whenever the contents are modified. Table 7 lists the notations. Specifically, we use *belief* to represent the in-memory and in-PM states of WOLVES, which are denoted as $\{\}_M$ and $\{\}_D$, respectively; we use *write()* to persist a container. We use before ($\ll$) and precedes ($\prec$) to indicate the ordering, for example, $A \ll B$ means A occurs before B, while $A \prec B$ means A occurs before B and holds until B occurs. Therefore, $A \prec B \Rightarrow A \ll B$.

A.2 Axioms

We reuse most rules in Sivathanu's logical framework [54] denoted by *S#*, as listed below.

- If a container B points to A in memory, its current generation also points to A in memory.

$$\{B^x \to A\}_M \Leftrightarrow \{g(B^x) \to A\}_M \tag{S1}$$

- If the on-disk contents of container A pertain to epoch y, some generation c should have pointed to generation $g(A^y)$ in memory followed by $write(A)$. The converse also holds:

$$\{A^y\}_D \Rightarrow \{c \to g(A^y)\}_M \prec write(A) \ll \{A^y\}_D \tag{S2}$$

$$\{c \to A_k\}_M \prec write(A) \Rightarrow \{A^y\}_D \wedge (g(A^y) = k) \tag{S3}$$

- If B points to A in memory, A is unshared and exclusive [54], a write of B will lead to the disk belief that B points to A:

$$\begin{aligned}\{B \to A\}_D \Rightarrow &\{B \to A\}_M \prec \\ &(write(B) \ll \{B \to A\}_D)\end{aligned} \tag{S4}$$

- A file system exhibits pointer ordering if it ensures that before writing a container B to disk, the file system writes all containers that are pointed to by B.

$$\begin{aligned}&\{B \to A\}_M \prec write(B) \\ &\Rightarrow \{B \to A\}_M \prec (write(A) \ll write(B))\end{aligned} \tag{S5}$$

We then present crucial axioms in WOLVES denoted by *W#* that govern the beliefs of WOLVES as follows.

- WOLVES believes that B logically points to A in PM only if B points to A physically with the correct checksum and no other entities are pointing to A with a larger timestamp.

$$\begin{aligned}\{B \to A\}_D \Leftrightarrow &\neg\forall C(\{C \dashrightarrow A\}_D \wedge ts(C) > ts(B)) \\ &\wedge \{B \dashrightarrow A\}_D \wedge ck(B)\end{aligned} \tag{W1}$$

- WOLVES ensures that if $\{B \to A_k\}$ and $\{C \to A_j\}$ hold in memory at two different points in time, container A should have been freed by persisting a newer C to invalidate B.

$$\begin{aligned}\{B \to A_k\}_M \ll \{C \to A_j\}_M \Rightarrow &\{B \to A_k\}_M \\ \ll &\{C \to A_j\}_D \wedge ts(C) > ts(B) \\ \ll &\{C \to A_j\}_M\end{aligned} \tag{W2}$$

Symbol	Description
$\&A$	set of entities that point to container A
A^x	the x^{th} version of epoch of A
A_k	the k^{th} version of generation of A
$g(A^x)$	the generation of the x^{th} epoch of A
$\{A^x\}_M$	the x^{th} version of A in the memory
$\{A^x\}_D$	the x^{th} version of A in the PM
$A \to B$	A logically points to B
$A \dashrightarrow B$	A physically points to B
$ts(A)$	the timestamp of container A (package)
$ck(A)$	the verification of checksum of A

Table 7: Notation used. *The table describes the basic symbols and operators used for consistency proof.*

- WOLVES ensures the data blocks A can be only pointed by one entity thanks to its CoW data update mechanism, which means:

$$\{B^x \to A^y\}_M \Rightarrow \{\&A = \{B^x\}\}_M \tag{W3}$$

A.3 Data Consistency

Data consistency ensures that a file will not contain data blocks belonging to another file [54]. Hence, we prove that if B points to A in PM (by scanning during recovery), then the generation of A is the same as the generation that was pointed to in memory, which can be reflected by the following rules:

$$\{B^x \to A_k\}_M \prec \{B^x \to A\}_D \wedge \{A^y\}_D \Rightarrow g(A^y) = k$$

We follow a similar process in §7.1 of [54]. We assume $g(A^y) \neq k$ (the content of A does not belong to generation k) and prove that this assumption leads to a contradiction, i.e.,

$$\neg(\{B^x \to A_k\}_M \prec \{B^x \to A\}_D \wedge \{A^y\}_D)$$

By applying eq. *S2*, we can obtain two possible sequences of the above assumption:

1. $\{B^x \to A_k\}_M \prec \{B^x \to A\}_D \ll \{C \to g(A^y)\}_M \prec write(A)$

2. $\{C \to g(A^y)\}_M \prec write(A) \ll \{B^x \to A_k\}_M \prec \{B^x \to A\}_D$

We then prove these two sequences lead to a contradiction.

- *Case 1*. Applying eq. *S1*,

$$\begin{aligned}\Rightarrow &\{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \\ &\ll \{C \to g(A^y)\}_M \prec write(A)\end{aligned}$$

Applying eq. *W2*, we have

$$\begin{aligned}\Rightarrow &\{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \ll \\ &(\{C \to g(A^y)\}_D \wedge ts(C) > ts(B)) \ll \{C \to g(A^y)\}_M \\ &\prec write(A) \\ \Rightarrow &(\{C \to g(A^y)\}_D \wedge ts(C) > ts(B)) \wedge \{A\}_D\end{aligned}$$

By applying eq. *W1*, it is easy to show that our initial assumption $\{B^x \to A\}_D$ (i.e., B points to A) and $\{C \to g(A^y)\}_D$ lead to a contradiction, as below.

$$\{B^x \to A\}_D \Rightarrow \neg\forall C(\{C \dashrightarrow A\}_D \wedge ts(C) > ts(B))$$

$$\{C \to g(A^y)\}_D \Rightarrow (\{C \dashrightarrow g(A^y)\}_D \wedge (ts(C) > ts(B)))$$

where $\neg\forall C(\{C \dashrightarrow A\}_D)$ means there is no entity C that points to A in PM, but $\{C \dashrightarrow g(A^y)\}_D$ means there is an entity C that points to $g(A^y)$ in PM.

- *Case 2*. Applying eq. *S1* and eq. *W2*,

$$\begin{aligned}
&\Rightarrow \{C \to g(A^y)\}_M \prec write(A) \\
&\quad \ll \{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \\
&\Rightarrow \{C \to g(A^y)\}_M \prec write(A) \\
&\quad \ll (\{g(B^x) \to A_k\}_D \wedge ts(B) > ts(C)) \\
&\quad \ll \{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D
\end{aligned}$$

Applying eq. *S4*,

$$\begin{aligned}
&\Rightarrow \{C \to g(A^y)\}_M \prec write(A) \\
&\quad \ll (\{g(B^x) \to A_k\}_D \wedge ts(B) > ts(C)) \\
&\quad \ll \{g(B^x) \to A_k\}_M \prec write(B) \ll \{B^x \to A\}_D
\end{aligned}$$

Applying eq. *S5* (as WOLVES obeys pointer ordering, *i.e.,* all the pointed data blocks/*packages* are written before the pointers are written),

$$\begin{aligned}
&\Rightarrow \{C \to g(A^y)\}_M \prec write(A) \\
&\quad \ll (\{g(B^x) \to A_k\}_D \wedge ts(B) > ts(C)) \\
&\quad \ll \{g(B^x) \to A_k\}_M \prec (write(A) \ll write(B)) \\
&\quad \ll \{B^x \to A\}_D
\end{aligned}$$

By eq. *S3*,

$$\begin{aligned}
&\Rightarrow \{C \to A\}_M \prec write(A) \\
&\quad \ll (\{g(B^x) \to A_k\}_D \wedge ts(B) > ts(C)) \\
&\quad \ll \{A^y\}_D \wedge (k = g(A^y)) \\
&\quad \ll write(B) \ll \{B^x \to A\}_D \\
&\Rightarrow \{A^y\}_D \wedge \{B^x \to A\}_D \wedge k = g(A^y)
\end{aligned}$$

This again leads to a contradiction since this implies that the contents of A on PM belong to the same generation A_k, while we started out with the assumption that $g(A^y) \neq k$.

A.4 Version Consistency

Version consistency ensures that the version of metadata x matches with the version of pointed data blocks y [15], which is a stricter consistency than data consistency. Given an in-PM beliefs $\{B^x \to A\}_D \wedge \{A^y\}_D$, we need to prove that there exists an in-memory belief $\{B^x \to A^y\}_M$ before the crash. This can be reflected by the following rules:

$$\{B^x \to A\}_D \wedge \{A^y\}_D \Rightarrow \{B^x \to A^y\}_M \ll \{B^x \to A\}_D \wedge \{A^y\}_D$$

To prove this, we follow a similar process in Appendix of [55]. If the L.H.S holds, by applying eq. *S4* and *S2*, we have:

$$\begin{aligned}
\{B^x \to A\}_D \wedge \{A^y\}_D \Rightarrow &(\{B^x \to A\}_M \prec write(B)) \\
&\wedge (\{C \to A^y\}_M \prec write(A))
\end{aligned}$$

This also implies two possible sequences and to prove the version consistency, we need to show that both sequences lead to $\{B^x \to A^y\}_M \ll \{B^x \to A\}_D \wedge \{A^y\}_D$.

1. $(\{B^x \to A\}_M \prec write(B)) \ll (\{C \to A^y\}_M \prec write(A))$

2. $(\{C \to A^y\}_M \prec write(A)) \ll (\{B^x \to A\}_M \prec write(B))$

- *Case 1*. We show that $A = A^y$ and $C = B^x$. If $A \neq A^y$, by applying eq. *S1* and eq. *W2*,

$$\begin{aligned}
&\Rightarrow (\{g(B^x) \to A\}_M \prec write(B)) \\
&\quad \ll (\{C \to A^y\}_M \prec write(A)) \\
&\Rightarrow (\{g(B^x) \to A\}_M \prec write(B)) \\
&\quad \ll (\{C \to A^y\}_D \wedge ts(C) > ts(B)) \\
&\quad \ll (\{C \to A^y\}_M \prec write(A)) \\
&\Rightarrow \{C \to A^y\}_D
\end{aligned}$$

This leads to a contradiction in our start assumption $\{B^x \to A\}_D$. Therefore, $A = A^y$. On the other hand, using eq. *W3*

$$\begin{aligned}
&\{B^x \to A^y\}_M \Rightarrow \{\&A = \{B^x\}\}_M \\
&\{C \to A^y\}_M \Rightarrow \{\&A = \{C\}\}_M \\
&\Rightarrow B^x = C
\end{aligned}$$

Hence,

$$(\{B^x \to A^y\}_M \prec write(B)) \ll (\{B^x \to A^y\}_M \prec write(A))$$

Hence, $\{B^x \to A^y\}_M \ll (\{B^x \to A\}_D \wedge \{A^y\}_D)$.

- *Case 2*. If $A \neq A^y$, by applying eq. *S5* to $(\{B^x \to A\}_M \prec write(B))$,

$$\begin{aligned}
&\Rightarrow \{B^x \to A\}_M \prec (write(A) \ll write(B)) \\
&\Rightarrow \{A^z\}_D \wedge z \neq y
\end{aligned}$$

This contradicts the start assumption $\{A^y\}_D$. Therefore, $A = A^y$. At the same time,

$$\begin{aligned}
&\Rightarrow \{B^x \to A^y\}_M \prec (write(A) \ll write(B)) \\
&\Rightarrow \{B^x \to A^y\}_M \prec (\{A^y\}_D \ll write(B)) \\
&\Rightarrow \{A^y\}_D \wedge \{B^x \to A\}_D
\end{aligned}$$

Thus, we have shown case 2 also leads to $\{B^x \to A^y\}_M \ll \{B^x \to A\}_D \wedge \{A^y\}_D$. Hence, WOLVES achieves version consistency.





Decentralized, Epoch-based F2FS Journaling with Fine-grained Crash Recovery

Yaotian Cui[1], Zhiqi Wang[1], Renhai Chen[2], Zili Shao[1]
[1]*The Chinese University of Hong Kong, China*
[2]*College of Intelligence and Computing, Tianjin University, China*

Abstract

F2FS, a log-structured filesystem, has gained widespread adoption in Android systems. However, F2FS relies on coarse-grained checkpointing for crash recovery. When triggered, this mechanism significantly degrades system performance by blocking file writes. Additionally, F2FS's checkpointing approach may not fully recover file data and metadata to a consistent state after a crash. Given these limitations, it is crucial to design a new journaling mechanism for F2FS that provides fine-grained crash recovery. While journaling methods are well-studied for in-place-update filesystems (such as JBD2 for EXT4), directly applying these state-of-the-art techniques to F2FS - an out-of-place-update filesystem - does not yield similar benefits.

In this paper, we propose a novel journaling technique, called F2FSJ, for F2FS with ordered journal mode. Catering to the out-of-place update features of F2FS, F2FSJ incorporate several innovative designs. First, in F2FSJ, only metadata changes are journaled and committed after data flushing, by which I/O and storage overheads can be mitigated. Second, we propose a decentralized journal design by embedding journal logs into inodes, which significantly reduces lock contention and interference when recording metadata changes. Third, we propose an epoch-based approach with a novel data/control-plane decoupling mechanism, which eliminates waiting times during journal period transfers. Finally, for journal apply, we propose a fast-forward-to-latest approach to consolidate multiple small updates into one update for reducing small writes. We have implemented a fully functional prototype of F2FSJ and conducted extensive experiments. Our experimental results demonstrate that F2FSJ can effectively reduce the checkpointing time by up to 4.9x and reduce the latency by up to 35% compared with F2FS. F2FSJ is open-sourced for public access.

1 Introduction

F2FS (Flash-Friendly File System) is widely adopted in Android systems due to its specially-tailored designs for flash storage such as append-only writes and hot/cold data separation with multi-head logging [10, 36, 39]. Despite the presence of battery power in Android smartphones, power failure remains a significant concern. For instance, Android smartphones may shut down in cold environments due to power-related issues [1, 13]. Furthermore, outdated applications or system updates can cause system freezes or crashes. As a result, ensuring effective crash recovery for F2FS is vital.

F2FS, however, relies on coarse-grained checkpointing for crash recovery, which once triggered, severely degrades the system performance by blocking file reads/writes, as all dirty data and metadata need to be written back to the disk as a checkpoint pack. For instance, as shown in §2.3, the worse-case latencies caused by checkpoints can be up to 293 ms, which is intolerable for user interaction or system responses. To address these issues, one effective approach is to utilize journaling that can mitigate these overheads by first recording file updates as logs and then applying logs to their original area. Thus, designing a new journaling mechanism for F2FS with fine-grained crash recovery is crucial.

Journaling mechanisms are well studied for in-place-update filesystems (e.g., JBD2 for EXT4 [7, 44, 45, 52, 53]). Nevertheless, directly applying these state-of-the-art journal methods for F2FS, which is an out-of-place-update filesystem, does not yield similar benefits. As illustrated in Figure 3, in an in-place-update filesystem, with fixed on-disk inode locations, in-memory changed inodes can be first journaled and then applied or recovered to on-disk inodes; by contrast, in F2FS, as inode locations are not fixed due to out-of-place-updates, by only journaling in-memory inodes, inodes cannot be correctly recovered due to outdated on-disk filesystem metadata. On the other hand, if both in-memory filesystem metadata and inodes are journaled, it incurs too much overhead as filesystem metadata and inodes need to be written twice.

In this paper, we propose a novel journaling technique, called F2FSJ, for F2FS with ordered journal mode. Ordered journal mode is widely adopted in practice by only committing changed metadata to the journal after data are flushed (each data flushing with its corresponding changed metadata is denoted as one journal period in this paper). To the best of

our knowledge, this is the first work to tackle F2FS ordered journaling.

In F2FSJ, we journal changes of filesystem and file metadata instead of the whole metadata pages employed in existing journal mechanisms such as JBD2. Specifically, F2FSJ includes several novel designs to address the following challenges: (1) How to organize a large volume of metadata changes so as to minimize lock contention with good scalability? (2) How to manage journal logging of each period and eliminate the waiting time when transferring from one journal period to another? (3) How to efficiently apply small changes recorded in the journal file and perform crash recovery with the change-based journaling for the out-of-place-update filesystem?

To address the first challenge, we propose a decentralized journal design by embedding journal logging into inodes. Specifically, for each inode, a per-inode log list is devised to record all of its related logging (i.e., file metadata changes from the inode and corresponding filesystem metadata changes from Segment Information Table (SIT), Node Address Table (NAT) and Segment Summary Area (SSA)). Compared with existing centralized approaches such as that employed by JBD2 in EXT4, our decentralized per-inode logging significantly reduces lock contention and interference among inodes when recording metadata changes, thus enabling higher efficiency and better scalability.

For the second challenge, we propose an epoch-based approach with a novel data/control-plane decoupling journal mechanism. Specifically, each journal period is associated with a new epoch; inside each epoch, journal's data and control planes are decoupled by logging metadata changes of an inode into its per-inode list (data plane) and only registering the inode information into the epoch (control plane). The data/control decoupling brings two benefits. First, it enables our decentralized journal design, i.e., for each new epoch, first storing metadata changes into per-inode lists and then collecting all changes of this journal period based on the registered inodes in the epoch for committing. Second, there is almost no waiting time when we transfer from one journal period to another. Particularly, when data flushing triggers the committing of the current epoch, we can immediately generate a new journal period with a new epoch to accommodate new metadata changes with negligible waiting time. On the contrary, the control and data planes are coupled with existing journal methods so the waiting time is inevitable during a period transfer (e.g., in JBD2, a new running transaction can only be issued after all file operations related to the current running transaction have been finished).

For the third challenge, when applying changes in the journal file, we adopt a fast-forward-to-latest approach to consolidate multiple small updates, caused by cross-epoch log records that operate on the same metadata, into one update, thus significantly reducing small writes. Specifically, we propose a new journal process design with a new page flag so a page with "dirty" metadata can be kept in memory without being flushed by the kernel page cache manager. Then when following the epoch order to apply changes, with a log record, we will directly flush its corresponding in-memory metadata and subsequently altering the page flag to "clean", thus avoiding future unnecessary flushing and enabling possible eviction as well. Note that our fast-forward-to-latest approach does not incur inconsistency due to the unique characteristic in out-of-place-update filesystems, i.e., old information will be kept without modification with out-of-place updates. Thus, in crash recovery, when there is no corresponding in-memory metadata when applying a log record, following the epoch order, metadata changes in log records can be applied one by one to the disk and finally reach to a consistent state.

We have implemented a fully functional prototype of F2FSJ and released the source code for public access. Our prototype involves minor changes (about 3,000 lines of C code) based on F2FS with Linux (Kernel Version 5.15). A series of experiments have been conducted to evaluate F2FSJ with benchmarks and real-world applications on both an Intel-based desktop and an ARM-based embedded development board. Experimental results show that F2FSJ can effectively reduce the checkpointing time by up to 4.9x and reduce the latency by up to 35% compared with F2FS. Additionally, F2FSJ, with its fine-grained crash recovery mechanism, can recover more files and data than F2FS checkpointing.

In summary, we make the following contributions:

- We for the first time propose a novel journaling technique for F2FS with ordered journal mode. F2FSJ eliminates the long checkpoint latency and provides fine-grained crash recovery.
- F2FSJ incorporates several innovative designs by leveraging the out-of-place-update features of F2FS as follows:
    1. F2FSJ only journals the changes of filesystem/file metadata so as to mitigate IO and storage overheads.
    2. In contrast to existing centralized approaches such as that employed by JBD2, a decentralized journal design is proposed to embed journal logs into inodes, which can significantly reduce lock contention and interference among inodes when recording metadata changes.
    3. An epoch-based approach with a novel data/control-plane decoupling journal mechanism is proposed to eliminate waiting times during journal period transfers that are inherent in existing methods (e.g., JBD2) with data/control coupling.
    4. A fast-forward-to-latest approach to consolidate multiple small updates into one update for reducing small writes in journal.

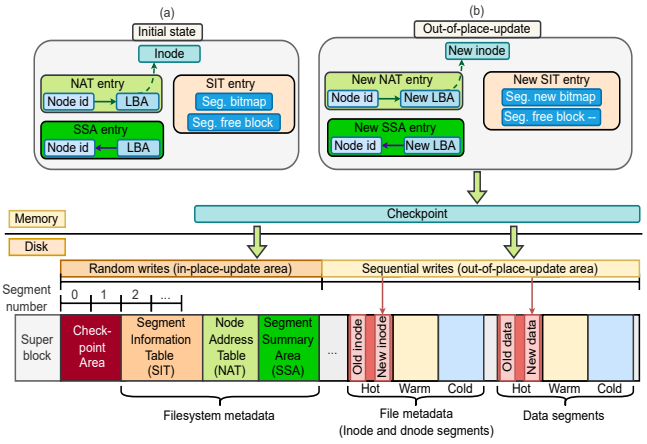


Figure 1: Overview of F2FS.

- We have implemented a fully functional prototype of F2FSJ in Linux kernel. The source code and detailed evaluation have been released for public access [8].

2 Background and Motivation

2.1 F2FS Overview

F2FS manages storage space at the unit of segments (e.g., 2MB per segment) and divides the whole space into the random writes area (in-place-update) and the sequential writes area (out-of-place-update). As shown in Figure 1, the random writes area stores F2FS filesystem metadata with fixed on-disk locations, such as SIT, NAT and SSA. Specifically, SIT records bitmap of each segment; NAT maintains the mapping from node IDs to logical block addresses (LBA) for node pages; SSA stores the number of valid blocks and the mapping from block addresses to node IDs. The sequential writes area stores file metadata and data in an append-only pattern. Specifically, inodes and dnodes (LBA of data blocks) are stored in node segments and data blocks are stored in data segments. F2FS separates node blocks and data blocks with three static temperatures (hot, warm, and cold) to exploit IO parallelism [28–30, 42] of multiple flash channels.

At runtime, on-disk filesystem metadata, file metadata and data are loaded into memory as illustrated in Figure 1(a). Then file operations can be performed. For instance, we can look up inodes by querying NAT and modify in-memory SIT and SSA for space management of each segment. In the sequential writes area, file metadata/data are managed using out-of-place updates. For instance, as illustrated in Figure 1(b), for regular file operations (e.g., file data writes) that modify file metadata and data, the updated inode is put to a new LBA rather than the original location.

In-memory and on-disk data/metadata will become inconsistent as they are first changed in memory. For instance, in Figure 1(b), to write the updated inode to a new LBA, a new LBA needs to be allocated, for which the in-memory NAT/SIT/SSA entries will be changed first, while the old versions are kept in the on-disk NAT/SIT/SSA. Such in-memory modifications will be lost if a crash happens before checkpointing.

2.2 Three Journaling Modes

In filesystems, there are typically three journaling modes, each offering a different level of data consistency and recovery capacity, as follows: (1) The first is ordered journal mode, in which only metadata are journaled and data flushing strictly precedes committing metadata journals. Ordered journal mode can ensure data integrity once metadata journals are committed to disk. (2) The second is writeback journal mode, which only journals metadata but the order of data flushing and metadata journals committing is not strictly enforced. Writeback mode can improve performance compared to the ordered mode since data can be written to disk quickly. However, it may introduce a risk of data inconsistency in a crash, as there is a possibility that the metadata journals may reach the disk before flushed data. (3) The third is data journal mode, in which journals both data and metadata. This mode ensures both data and metadata consistency during recovery. However, it has a performance impact due to the additional overhead of journaling data. In this paper, we focus on ordered journaling mode, which is widely adopted in practice (e.g., it is the default journaling mode in EXT4 [24]).

2.3 F2FS Checkpointing and Problem

F2FS utilizes checkpointing for crash recovery. Checkpoints can be triggered by either the thresholds of dirty in-memory metadata (e.g., the cached NAT entries or dirty node pages exceed a setting ratio) or a timeout (e.g., 60 seconds by default). When a checkpoint is triggered, all write activities will be blocked until it is completed. During a checkpoint, in-memory file data, file metadata and filesystem metadata (i.e., NAT, SIT, and SSA) will be flushed to disk in order to form a checkpoint pack that is stored in the on-disk checkpoint area as a snapshot. Based on F2FS checkpointing, for crash recovery, F2FS can roll back to a consistent checkpoint with its roll-back recovery mechanism. There are three critical issues associated with F2FS checkpointing.

Problem 1: Time Overhead. We have conducted preliminary experiments to reveal the overheads with the experimental setup in §5.1. In the experiments, four benchmarks, `mkdir`, `rmdir`, `create-4KB` and `unlink-4KB`, from filebench [9], are utilized, which all contain intensive file operations on filesystem metadata and file metadata, and the total execution time, the checkpoint time are measured. Figure 2(a) shows the time breakdown of the average execution time for F2FS and F2FSJ. It can be observed that with F2FS, as checkpoints are frequently triggered with intensive filesystem/file metadata operations, the checkpoint time occupies a significant proportion of the total execution time, ranging from 37.2%, 17.2%, 47.3% to 44.2%. By contrast, F2FSJ, with its fine-grained journaling, can significantly reduce the total execution time

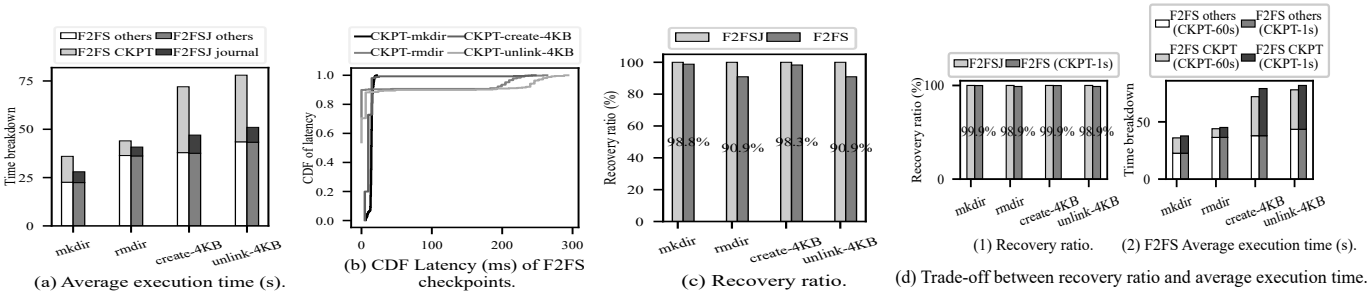


Figure 2: Preliminary experiments with the four benchmarks (mkdir, rmdir, create-4KB, unlink-4KB) from filebench, which contain intensive filesystem/file metadata operations. (a) Time breakdown of the average execution time for F2FS and F2FSJ. (b) CDF latencies of the F2FS checkpoints. (c) Recovery ratio comparison of F2FS and F2FSJ. (d) Trade-off between recovery ratio and average execution time: (1) Recovery ratio comparison of F2FS with the one-second checkpointing interval (denote as CKPT-1s) and F2FSJ; (2) Average execution time comparison of F2FS with the sixty-second and one-second checkpointing intervals (denote as CKPT-60s and CKPT-1s, respectively).

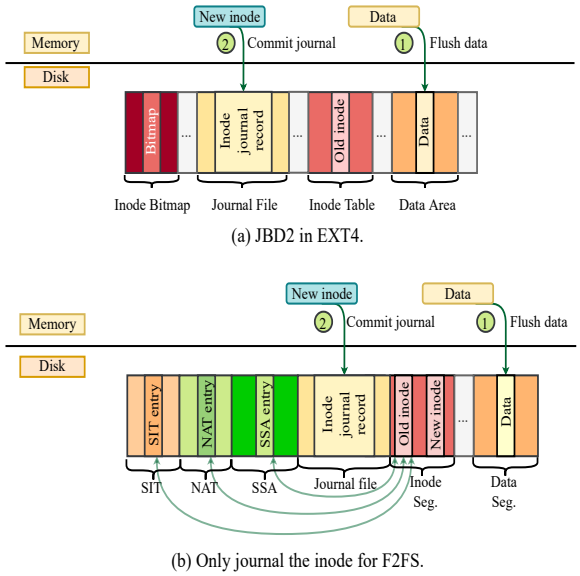


Figure 3: Journaling in EXT4 and F2FS. (a) In EXT4, as on-disk inode locations are fixed, a changed inode can be first stored in the journal file and then applied or recovered to on-disk inode. (b) In F2FS, inode locations are not fxed due to out-of-place-updates, so by only journaling in-memory inodes, inodes cannot be correctly recovered due to outdated on-disk fliesystem metadata.

across the board.

Another critical issue for F2FS checkpointing is its long tail latency. Figure 2(b) shows the CDF (Cumulative Distribution Function) latencies of the F2FS checkpoints. For instance, the worst-case latency reaches to 247ms, 233ms, 293ms, 26ms for `create-4KB`, `rmdir`, `unlink-4KB` and `mkdir`, respectively, which is intolerable for user interaction or system responses.

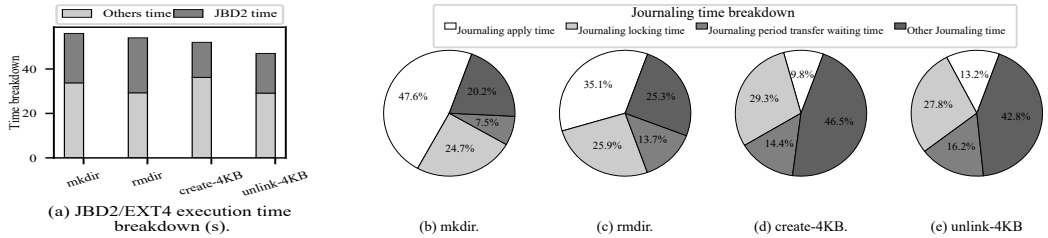
Problem 2: Data and Metadata Loss. F2FS checkpointing is triggered by either the thresholds of dirty in-memory metadata or a timeout (e.g., 60 seconds by default), so only coarse-grained crash recovery can be performed and newly modified metadata and data are lost in a crash between two checkpoints. Note that in F2FS, `fsync()` will not trigger checkpoints for performance efficiency, so flushed file data could be lost. To demonstrate file data/metadata loss, we have conducted crash experiments for F2FS checkpointing and F2FSJ. Specifically, for F2FS, a sudden power failure is made after the last checkpoint is triggered, and for F2FSJ, at the same crash point, a power failure is made with `poweroff` [18]. Figure 2(c) shows the recovery ratio comparison of F2FS checkpointing and F2FSJ. It can be observed that F2FS with the default checkpointing interval (i.e., 60 seconds) incurs more data/metadata loss (up to 9.1%) compared with F2FSJ.

A simple solution to reduce data/metadata loss is to shorten the checkpointing interval, which, however, is a trade-off between metadata/data loss and performance degradation. As shown in Figure 2(d), when the checkpointing interval is set to one second, F2FS has less data/metadata loss (up to 1.1%). Nevertheless, it comes at the cost of the increased checkpointing execution time, i.e., the average execution time of checkpointing increases 12.2%, 14.5%, 21.2%, and 11.0% for the four benchmarks, respectively.

Problem 3: Inconsistency with Roll-forward Recovery. In F2FS, any persistent data or metadata beyond a checkpoint cannot be recovered in the event of a crash using its roll-back recovery mechanism. To address this limitation, F2FS implements a roll-forward recovery mechanism [4, 14, 15], which tags inodes and dnodes in fsync and utilizes this tag information for data and metadata recovery. However, in no-barrier and POSIX fsync modes, the unordered nature of the I/O stack [16, 31, 40, 55] can lead to inconsistencies, as the persistence order of file data and its associated inode/dnode tags is not guaranteed. For instance, if the inode/dnode tags are successfully written while the corresponding file data fails to persist due to reordering, the recovered state may become inconsistent - resulting in a newer inode pointing to outdated data.

2.4 Motivaton

To address the above issues of F2FS checkpointing, we propose F2FSJ that is a journaling technique with fine-grained crash recovery. Next, we first present the problems with existing journal methods in designing F2FS journaling and then introduce new challenges for applying our metadata-change-



(a) JBD2/EXT4 execution time breakdown (s). (b) mkdir. (c) rmdir. (d) create-4KB. (e) unlink-4KB

Figure 4: Preliminary experiments with the four benchmarks (mkdir, rmdir, create-4KB, unlink-4KB) from filebench for comparing the execution time with time breakdown on JBD2 in EXT4 (a), and journaling time breakdown for (b) mkdir, (c) rmdir, (d) create-4KB, and (e) unlink-4KB.

based journaling method in F2FSJ.

2.4.1 Problems with Existing Journal Methods

While journaling methods are well-studied for in-place-update filesystems (such as JBD2 for EXT4) [35, 37, 41, 47, 48, 51, 54], directly applying these state-of-the-art techniques to F2FS does not yield similar benefits. Specifically, as illustrated in Figure 3(a), for EXT4, as an in-place-update flesystem, its on-disk inode locations are fixed, so a changed inode in memory can be first journaled and then correctly applied or recovered to on-disk inodes without filesystem metadata modification. By contrast, for F2FS, as its inodes and data are managed using out-of-place updates, inode locations are not fixed. Thus, as shown in Figure 3(b), by only journaling the inode, it cannot be correctly recovered as the corresponding on-disk filesystem metadata (e.g., SIT/NAT/SSA entries) are outdated. On the other hand, if we store both in-memory filesystem metadata and inodes in the journal file, it incurs more overheads than JBD2, since both filesystem metadata and inodes need to be written twice.

Another problem in existing journal methods is to journal changed metadata based on pages (blocks). For instance, even if only the file size information is updated in an inode, JBD2 will journal the whole 4KB page that stores the inode. This incurs both runtime and storage overheads. First, at runtime, subsequent file operations that operates on the page will be held up when a journal commit involving that page occurs. As shown in Figure 4 (a), the average execution time of JBD2 accounts for a significant portion, ranging from 39.7%, 45.7%, 30.3%, to 38.8% for the four benchmarks, respectively. Second, not only more space is occupied in journal files, but also garbage collection will be triggered earlier, during which journal commit will be blocked so extra runtime overheads are introduced. In the journaling, time breakdowns are shown in Figures 4(b), 4 (c), 4(d), 4(e), it can be observed that JBD2 apply, which operates on journal files, occupies a big portion of the total journaling time, e.g., about 47.6% and 35.1% in benchmarks `mkdir` and `rmdir`, respectively.

2.4.2 Challenges for Journaling Metadata Changes

Motivated by §2.4.1, in F2FSJ, we propose a metadata-change-based journaling policy, in which we only journal the changed part instead of the whole page for in-memory metadata. However, designing F2FS journaling with such a policy poses new challenges as follows: (1) Lock contention; (2) Long waiting times during journal period transfers.

Challenge1: Intensive Locking. Existing journal methods utilize a centralized approach that leads to intensive locking contention. For instance, such an approach is employed by JBD2, in which a transaction is used to represent a journal period, and a global journal ticket (`t_updates`) and a log list is maintained for journaling management. Once a file operation triggers data or metadata to be journaled, it must first obtain a journal ticket, and then add the data/metadata pages into the log list of the running transaction. Correspondingly, two locks are provided for mutually exclusive accesses for the journal ticket and the journal log list, respectively. Lock contention for them becomes one of the performance bottlenecks in JBD2 [53]. Figures 4(b), 4 (c), 4 (d), 4(e) show the time breakdown of the locking overheads, in which (1) `j_state_lock` is the lock contention time for the journal ticket; (2) `j_list_lock` is the lock contention time for the log list. It can be observed that such lock contention occupies 24.7%, 25.9%, 29.3%, 27.8% of the total journaling time for the four benchmarks, respectively. Instead, in F2FSJ, we adopt a decentralized approach by storing logs into each inode (i.e., via per-inode log lists), thus significantly mitigating lock contention and interference among inodes.

Challenge2: Long waiting Times During Journal Period Transfers. Existing journal designs like JBD2 couples the control and data planes, thus incurring long waiting times during journal period transfers. For instance, in JBD2, a new running transaction cannot be issued when the previous one starts to commit. The reason is that for each transaction, the data plane (the log list) and the control plane (e.g., journal ticket, journal committing, etc.) are coupled, so a new transaction (for generating new data in the data plane) must wait for the completion of the previous one (for committing in the control plane). As shown in Figures 4(b), 4(c), 4(d), 4(e), in JBD2, the waiting time for the journal period transfers occupies about 7.5%, 13.7%, 14.4%, 16.2% of the total journaling time, respectively. In F2FSJ, this is solved by our epoch-based approach with data/control-plane decoupling.

3 F2FSJ Design

3.1 Overview

F2FSJ incorporates several novel designs. First, we employ epochs to manage control plane journaling, which can

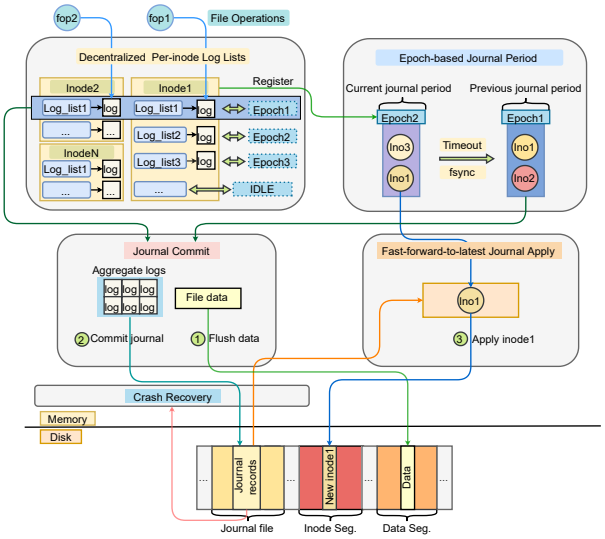


Figure 5: Workflow Illustration of F2FSJ.

effectively represent a journal period and decouple the control/data planes of journaling. Second, we adopt decentralized per-inode log lists to manage journaling data plane, by which metadata changes caused by file operations are inserted into corresponding per-inode log lists without intensive locking competition and inter-inode interference. Third, we propose an asynchronous journal commit mechanism that does not block ongoing file operations modifying the same metadata. Finally, we propose a fast-forward-to-latest journal applying scheme that can consolidate multiple small updates into one update for reducing small writes in journal apply.

Figure 5 illustrates the workflow of F2FSJ. First, when file operations modify metadata, metadata changes are inserted into inode log lists as log records and the inode information related to the changes is registered into the epoch of the current journal period (e.g., Epoch2 in Figure 5). Second, when an fsync or timeout is invoked, a new journal period with a new epoch (e.g., Epoch2) is generated to accommodate new metadata changes, while the changes can be aggregated based on the inode information in the previous epoch (e.g., Epoch1) and then committed after data flushing. Third, on-disk journals can be applied in epoch order, in which we adopt a fast-forward-to-latest approach by directly flushing in-memory metadata pages for the same metadata in logs. Finally, for crash recovery, we can replay on-disk journal records and reach to a consistent state.

3.2 Epoch-based Journaling

We use epochs to manage control plane journaling, and an epoch is represented with an ID that is a monotonically increasing number. Specifically, when an `fsync` or timeout is invoked, we will transfer to a new journal period with a new epoch and then the previous epoch can enter the commit phase. With a new epoch, for an inode, when its metadata changes occur for the first time in this epoch, its inode number is registered into the epoch (on the other side, in the inode,

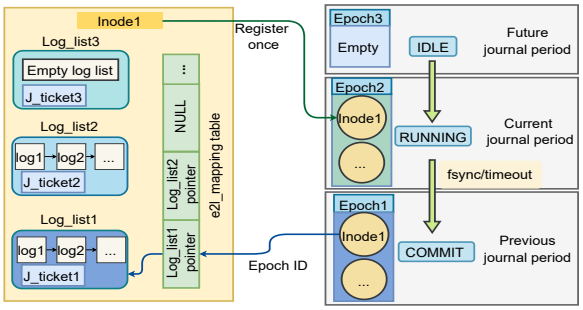


Figure 6: Illustration of epoch-based journaling and per-inode log lists.

based on the `e2l_mapping` table, with the epoch ID, the log list that stores metadata changes can be found as discussed in Section §3.3). In our implementation, inside an epoch, we utilize a linked list to record inode numbers and a lock is employed to ensure atomic inode registration. Note that for one epoch, inodes only need to register once (when its metdata changes occurs for the first time in this epoch). Thus, the lock contention overhead is negligible.

To align epochs with journal phases, each epoch has three states: (1) `IDLE` signifies availability for future journal periods without registered inodes; (2) `RUNNING` indicates that the epoch is being used for the current journal period; (3) `COMMIT` signifies that the epoch has entered the journal commit state. Specifically, as illustrated in Figure 6, each epoch maintains inode information, and the same inode can be registered in different epochs if it is involved in multiple journal periods. After inode registration, the logs can be efficiently stored in decentralized per-inode log lists. When an epoch is triggered to commit by `fsync` or timeout, it stops registering new inodes and transitions to the `COMMIT` state. Simultaneously, a new epoch is immediately issued for the next journal period, thus incurring negligible waiting times for journal period transfers.

There are three benefits of utilizing epoch. First, by decoupling the data/control planes, an epoch only records the inodes involved in journaling, while metadata changes are stored separately in the data plane (i.e., per-inode log lists, as discussed in Section §3.3). With such a design, lock contention can be effectively reduced due to the small-sized and simplified control plane. Second, based on the decoupled control/data planes, the new epoch for the next journal period can be issued immediately without having to wait for ongoing file operations belonging to the current epoch. Third, the ACID property [25] can be ensured in journal committing as the logs belonging to the same epoch can be committed as a single transaction.

3.3 Decentralized Per-inode Log Lists

In F2FSJ, decentralized per-inode log lists are utilized to efficiently manage journaling data plane. Specifically, when file operations modify the metadata of an inode (e.g., the file size), based on the modified parts (i.e., metadata changes), log records are generated and inserted into the corresponding log list in the inode. An inode may contain multiple log

lists that are associated with different epochs, but only one log list is corresponding to the RUNNING epoch. Inside an inode, an epoch-to-log-list mapping table (i.e., `e2l_mapping`) is maintained, so for a given epoch ID, its corresponding log list can be found, as illustrated in Figure 6.

Based on `e2l_mapping` tables, we can do inode registration and store metadata changes as log records in per-inode log lists. Specifically, when inserting log records into an inode, based on the epoch ID of the RUNNING epoch, we first check the `e2l_mapping` table of the inode. If this is the first time to insert log records for this inode in this epoch, `NULL` will be returned. In this case, we first register the inode number into the RUNNING epoch, and then create a new log list (e.g., implemented by a linked list) and put the list address (e.g., the head of the linked list) into the `e2l_mapping` table, after which the log records can be inserted into the log list accordingly. Otherwise, if we can find the address of the log list from the `e2l_mapping` table, log records can be directly inserted into the list.

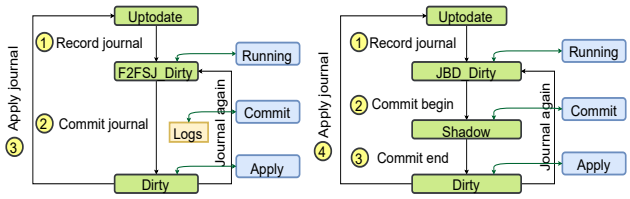
In F2FSJ, for filesystem metadata such as SIT, NAT and SSA, their changes are stored into per-inode log lists as well. Specifically, when collecting and storing metadata changes of an inode, if they are related to file data (e.g., data allocation, data deletion, etc.), the fliesystem metadata changes associated with the file data (i.e., SIT and SSA entries) will be recorded into the log list by associating with the corresponding metadata changes of the inode. On the other hand, the filesystem metadata of an inode (i.e., NAT, SIT and SSA entries of the inode) will be appended to journal records during journal commit (discussed in Section §3.5). As such, in journal apply and crash recovery, with both file and filesystem metadata in journal records, we can reach to a consistent state.

During epoch transition (from `RUNNING` to `COMMIT`), to determine whether file operations have completed journaling, a counter (e.g., `J_ticket` as shown in Figure 6) is associated with each log list, which represents the number of ongoing file operations. The commit thread can only aggregate the logs of a log list when the counter equals to zero.

Our design with decentralized per-inode log lists brings several benefits. First, it is convenient for log collection because each inode can naturally collect its own logs (i.e., metadata changes) in file-operation paths. Second, with our decentralized design, metadata changes are distributed to different inodes, thus eliminating the need for global locks in journaling data plane and significantly reducing lock contention and inter-inode interference. Third, the memory overhead is minimized as only metadata changes are journaled, and log lists are created on demand (only when metadata changes occur for inodes in epochs) and reclaimed after journal commit.

3.4 Non-blocking Journal Commit

`COMMIT` epochs are committed based on their commit order. Specifically, a journal commit process has two phases, namely, log aggregation and journal committing. In the first phase for log aggregation, log records are aggregated from the log list of each inode in the epoch for subsequent journal writes. Specifically, for each inode registered in the epoch, the commit thread locates the corresponding log list through `e2l_mapping` using the `COMMIT` epoch ID. Subsequently, the logs are aggregated together by iterating through the log list. Finally, the SIT/NAT/SSA entries of this inode will be appended to the end of the aggregated logs.



(a) Page state changes in F2FSJ. (b) Page state changes in JBD2/EXT4.

Figure 7: Journal process with page state changes.

The second phase is journal committing, by which aggregated logs in the first phase are written into an on-disk journal file with the following three steps. First, a journal descriptor block (JD) is written, which is a journal header that contains the metadata such as the epoch ID and the log-record length. Second, aggregated logs are persisted. Finally, a journal commit block (JC) is persisted to denote the completion of the journal committing. This three-step process can guarantee that all logs of one epoch can be applied or restored atomically.

In F2FSJ, we can provide non-blocking journal commit, meaning that journal committing does not block ongoing file operations that operate on the same metadata, since only metadata changes in log lists are journaled. By contrast, in JBD2, journaling is based on metadata pages that are locked during the DMA transfer in journal commit, so ongoing file operations on these metadata pages will be blocked.

3.5 Journal Apply

For journal apply, we adopt a fast-forward-to-latest approach, which can consolidate multiple small updates, caused by cross-epoch log records that operate on the same metadata, into one update. The key idea is to directly flush corresponding in-memory metadata instead of applying log records one by one. It consists of two components: (1) Journal process with page state changes, and (2) Fast-forward-to-latest journal apply, as follows.

Journal Process with Page State Changes. We propose a new journal process design with a new page flag so a page with "dirty" metadata can be kept in memory without being flushed by the kernel page cache manager. Specifically, in F2FSJ, each in-memory page has three states: (1) `Uptodate` that represents this page has the same content as its on-disk counterpart; (2) `F2FSJ_Dirty` that represents this page has been modified and the changes have been recorded but not committed, while the data has not been flushed to disk; and (3) `Dirty` that represents this page has been modified and the changes have been committed but not applied, while the corresponding data has been flushed. The state changes are

illustrated in in Figure 7(a). Particularly, the kernel page cache manger can only evict `Uptodate` pages, while it cannot do `Dirty` or `F2FSJ_Dirty` pages. During journal apply, `Dirty` pages will be utilized for reducing small writes as shown next.

For ordered journal mode, data should be persisted before metadata. For `F2FSJ_Dirty` metadata pages, their data have not been flushed, while after journal commit that happens after data flushing, metadata pages are changed to `Dirty`. Thus, `Dirty` metadata pages can be used for journal apply. Compared with JBD2, which uses two page states `JBD_Dirty` (before journal commit) and `Shadow` (during journal commit for preventing ongoing file operations from modifying the committed metadata pages) as shown in Figure 7(b), F2FSJ only requires one state (i.e., `F2FSJ_Dirty`), thus simplifying the design and implementation in journal apply.

Fast-forward-to-latest Journal Apply. In journal apply, following the commit order, for each journal record, by checking its corresponding in-memory metadata page, we perform as follows: (1) If the page is found in memory and its state is `Dirty`: we directly flush the `Dirty` page and subsequently altering its state to `Uptodate`; (2) If the page is not found in memory or the page state is `Uptodate`, we skip this record (based on our state changes, only `Uptodate` pages can be evicted so this is the case that its corresponding `Dirty` page has been flushed to disk); (3) If the page state is `F2FSJ_Dirty`, we apply the journal record by updating its on-disk data accordingly. Note that in journal apply, after an inode is updated, the filesystem metadata of the inode (i.e. SIT, NAT and SSA) may need to be updated as shown in Figure 8. Conceptually, only two states - `Dirty` and `Uptodate` - may seem sufficient. However, an additional state, `F2FSJ_Dirty`, is essential for implementing the fast-forward-to-latest approach. This state specifically indicates that an in-memory metadata page associated with a journal record is also involved in committing another ongoing journal record, making it unavailable for use. In such cases, the journal record in the journal file is applied directly, with the file data and metadata appended in F2FS.

An example is given in Figure 8. In Figure 8(a), we apply epoch1's journal record as follows: (1) Read the epoch1's journal record to memory and find that one log record of inode1 (i.e., 4KB file size) needs to be applied; (2) Use the inode number recorded in the log to find the in-memory `Dirty` inode1 (with 8KB file size); (3) Apply the in-memory `Dirty` inode1 to a new location to disk and change the page state of the in-memory inode1 to `Uptodate`; and (4) Update the corresponding NAT entry and write the new NAT entry to disk. In Figure 8(b), after applying epoch1's journal, we apply epoch2's journal record, in which after we read epoch2's journal to memory and find that one log record of inode1 (8KB file size) needs to be applied, we use the inode number to locate the in-memory inode1 and find its state is `Uptodate`, then we skip applying this log and move to next epoch.

Journal apply is conducted following the commit order. Specifically, we always start with the current epoch indicated

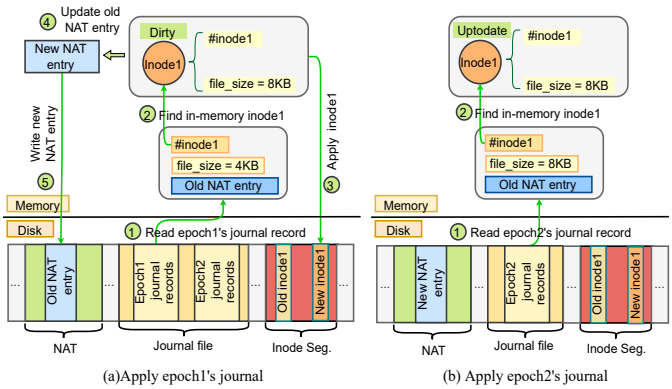


Figure 8: Illustration of the fast-forward-to-latest journal apply.

by the starting offset of the journal file that is recorded on disk. Only after all journal records of the current epoch have been applied, the offset can be updated so we can move to next epoch. Following this, with the unique characteristic in out-of-place-update file systems, i.e., old information will be kept without modification with out-of-place updates (e.g., Old inode1 is not replaced by New inode1 as shown in Figure 8), the fast-forward-to-latest approach does not incur inconsistency even with system crash. Journal apply is triggered and stopped based on the low and high watermarks based on the free space of the journal file. With a normal system shutdown, all journal records in the journal file will be applied before the close.

3.6 Crash Recovery

Crash recovery will be performed when we find there are journal records that are not applied during system booting. Similar to journal apply, we read journal records from the journal file and apply them one by one. However, in contrast to journal apply, we can directly apply journal records without checking in-memory pages. Specifically, based on the filesystem metadata of an inode (i.e., the SIT/NAT/SSA entries) recorded in the journal file, we can locate and read the old inode to memory, apply metadata changes based on logs and then update the new inode and its SIT/NAT/SSA entries to disk.

In the event of a crash during journal apply, the out-of-place-update mechanism in F2FS ensures that old data remains intact. This feature enables the filesystem to recover by replaying on-disk journals in epoch order, thereby maintaining data consistency and integrity. For instance, as illustrated in Figure 8(b), if a system crash occurs during epoch2, recovery will resume from epoch2. Using the filesystem and file metadata from epoch2's journal records, the system can then restore a consistent state.

4 Implementation

We have implemented F2FSJ with minor modifications to F2FS with about 3,000 lines of code, summarized as follows.

First, we define various log types and structures for different file operations, which contain the type of file operations

Categories	Name	Description
Metadata-intensive workloads with Filebench [9]	mkdir	Create 4M directories.
	rmdir	Delete 4M directories.
	create-empty	Create 4M empty files.
	unlink-empty	Delete 4M empty files.
Small-file-intensive workloads with Filebench	create-4KB	Create 4M 4KB-files.
	unlink-4KB	Delete 4M 4KB-files.
	copy-4KB	Create and copy 2M 4KB-files.
Data-intensive workloads for a 32GB-file with FIO [11]	Seq_write	Sequentially write.
	Rand_write	Randomly Write.
	Seq_read	Sequentially read.
	Rand_read	Randomly read.
	SeqWR_W	Write bandwidth of mixed Sequential writes/reads.
	SeqWR_R	Read bandwidth of mixed Sequential writes/reads.
	RanWR_W	Write bandwidth of mixed random write/reads.
	RanWR_R	Read bandwidth of mixed random write/reads.
Realistic applications with Filebench and APP commands	Varmail	Frequently create and delete small files with fsync.
	OLTP	Intensively write and append data to files.
	Fileserver	Intensively write and read small files.
	Webserver	Intensively read small files with fewer writes.
	Webproxy	Intensively read small files with fewer writes and deletions.
	git clone	Download a remote 4.7GB code repository.
	cp -r	Copy a local 3.9GB linux source code.
	make Linux	Compile Linux with local source code.
Android workloads with Mobibench [21]	Twitter	Replay I/O trace of Twitter.
	Facebook	Replay I/O trace of Facebook.
	SQlite	SQlite performance with varying journal mode.

Table 1: Benchmarks introduction.

and the contents of metadata changes. We implement interfaces for logging and integrate them into file operations.

Second, we modify `struct f2fs_inode_info` to implement per-inode log lists and `e2l_mapping` mapping tables. Specifically, for each log list, we set a journal ticket counter to record ongoing file operations. We also use an array to implement the `e2l_mapping` table and implement epochs with linked lists.

Third, we implement a kernel thread for journal commit that has a linked list for `COMMIT` epochs. We use another kernel thread for journal apply, which is triggered and stopped based on the low and high watermarks of the free journal space. Considering garbage collection (GC) may migrate inodes and data, thus changing the locations of old inodes, all journal records need to be applied before GC.

Finally, F2FSJ requires a contiguous disk area to store journal records, for which an on-disk journal file with a configurable size (e.g., 256 MB in our experiments) is created and utilized in round-robin fashion. During each epoch, metadata changes are collectively appended to the journal file, effectively avoiding random writes. Furthermore, by journaling only metadata changes instead of entire metadata pages, it significantly reduces write amplification.

5 Evaluation

In this section, we present evaluation results on F2FSJ and seek to answer the following questions: (1) Does F2FSJ eliminate the long tail latency of F2FS checkpointing? (2) Does F2FSJ provide better performance under various workloads than F2FS? (3) Does F2FSJ gain better performance compared with JBD2? (4) What are the overheads of memory, CPU and storage in F2FSJ?

5.1 Experiment Setup

Experiment Environment. All the experiments are performed on a desktop machine equipped with Intel's i9-10850K CPU (19 cores running at 3.6 GHz), 64 GB DRAM and a

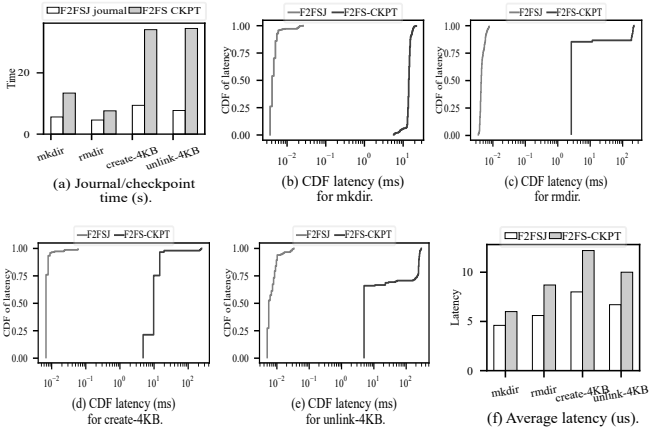


Figure 9: Journaling and checkpointing time comparison with filebench: (a) The journal time of F2FSJ versus the checkpoint time of F2FS checkpointing. (b)-(e) CDF latency comparison between F2FSJ and F2FS checkpointing for mkdir, rmdir, create-4KB and unlink-4KB. (f) Average latency comparison between F2FSJ and F2FS checkpointing for mkdir, rmdir, create-4KB and unlink-4KB.

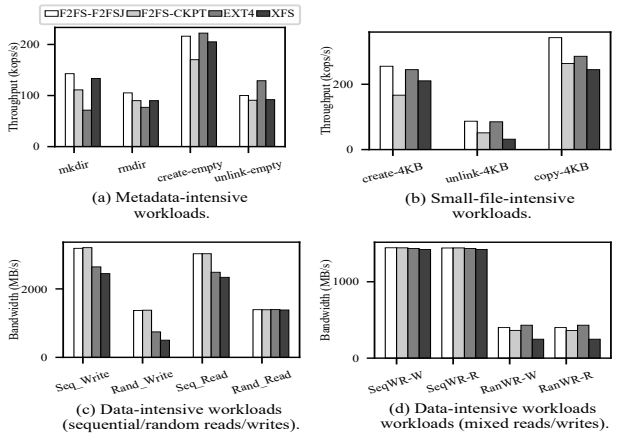


Figure 10: End-to-end performance: Throughput comparison with filebench for (a) Metadata-intensive workloads and (b) Small-file-intensive workloads. Bandwidth comparison with FIO for (c) Data-intensive workloads and (d) Data-intensive workloads with mixed reads/writes.

256GB Samsung 980pro PCIe 4.0 NVMe SSD [27, 57] that can provide up to 7.0 GB/s and 5.0 GB/s throughput for sequential reads and sequential writes, respectively. Clearlinux with Linux kernel v5.15.39 [20] is used as operating system.

Experiment Benchmarks. As shown in Table 1, we categorize the benchmarks into five classes, all of which have different file and directory access patterns, ranging from metadata-intensive workloads, small-file-intensive workloads, data-intensive workloads, realistic applications and Android workloads.

For evaluation results, all the tests are started with a clean filesystem and all reported numbers are the mean of at least ten runs and the standard deviation in all cases is less than 5% of the mean. We use `flame` [12], `perf` [22], and `bcc-tools` (eBPF based) [3], `Top` [19] and `Free` [17] to measure and breakdown the experiment results.

5.2 Journal and Checkpoint Time Comparison

We begin by comparing journal time and checkpoint time. In this context, journal time in F2FSJ refers to the execution duration of journaling for a specific benchmark, while checkpoint time in F2FS checkpointing represents the execution duration of the checkpointing process. Next, we evaluate latency, where latency indicates the execution time of a single operation, and average latency reflects the mean execution time of all such operations within a given benchmark.

We use four filebench benchmarks: `mkdir`, `rmdir`, `create-4KB`, `unlink-4KB`. As shown in Figure 9(a), F2FSJ has the minimal time overhead. For instance, the checkpoint time with F2FS checkpointing is 2.4x, 1.7x, 3.6x, and 4.9x longer than the journaling time with F2FSJ across the four benchmarks. Additionally, in Figures 9(b)-9(e), F2FSJ significantly reduces tail latencies compared to F2FS checkpointing, with the latter exhibiting tail latencies that are three orders of magnitude higher. Finally, as shown in Figure 9(f), F2FSJ outperforms F2FS checkpointing in terms of the average latency (e.g., 23%, 35%, 13% and 33% reductions for `mkdir`, `rmdir`, `create-4KB` and `unlink-4KB`, respectively).

5.3 End-to-end Performance

In this section, we use a diverse set of workloads, along with the throughput (operations per second) and IO bandwidth (MB/second) metrics, to assess and compare the end-to-end performance of multiple filesystem combinations. These combinations include F2FS integrated with F2FSJ (referred to as F2FS-F2FSJ), F2FS utilizing its checkpointing mechanism (denoted as F2FS-CKPT), EXT4 in conjunction with JBD2 (simply called EXT4), and XFS with logical journaling (simply named XFS). In addition, filesystems are set to the default working mode. Specifically, F2FS operates in POSIX mode, EXT4 with JBD2 runs in ordered mode, and XFS is configured with metadata journaling and asynchronous transactions. We first evaluate end-to-end performance using micro-benchmarks (§5.3.1) and real applications (§5.3.2). We then present the results for scalability (§5.3.3) and long-term/aged filesystem testing (§5.3.4).

5.3.1 Results with Micro-benchmarks

Metadata-intensive workloads. Figure 10(a) presents the throughput results. F2FS-F2FSJ outperforms F2FS-CKPT by 1.29x, 1.16x, 1.27x, 1.11x on `mkdir`, `rmdir`, `create-empty` and `unlink-empty`, respectively. For `mkdir` and `rmdir`, F2FS-F2FSJ (2.0x and 1.37x higher throughtput) and F2FS-CKPT (1.56x and 1.17x higher throughput) have better performance than EXT4. This is because EXT4 requires journaling 6 metadata pages for each create or delete operation, leading to more IO operations with more storage space. Additionally, file operations need to wait for JBD2 checkpoint in order to recycle space. But for `create-empty` and `unlink-empty`, EXT4 outperforms F2FS-F2FSJ (1.03x and 1.29x), because the longer query time on NAT is required in F2FS-F2FSJ and file operations will not be blocked by JBD2 checkpoint in EXT4. On the other hand, F2FS-F2FSJ outperforms XFS (i.e., 1.05x-1.17x higher throughput) for all benchmarks.

Name	Avg. file size	Num. of files	R/W/C/U ratio	Total file operations
Varmail	16 KB	1 M	2:2:1:1	128.8 K
OLTP	10 MB	3.2 k	1:10:0:0	33 M
Fileserver	128 KB	800 k	1:2:1:1	1.1 M
Webserver	16 KB	1.6 M	10:1:0:0	12.4 M
Webproxy	16 KB	2 M	5:1:1:1	3.8 M

Table 2: Filebench realistic workloads parameters. R/W/C/U represents the ratio of read, write, create and unlink, respectively.

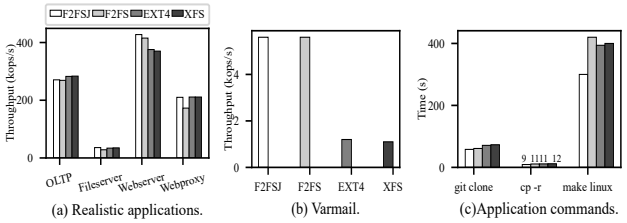


Figure 11: Throughput comparison with filebench for (a) Realistic applications and (b) Varmail. (c) Time comparison with APP commands.

Small-file-intensive workloads. As shown in Figure 10 (b), F2FS-F2FSJ outperforms F2FS-CKPT by 1.14x, 1.69x and 1.30x on `create-4KB`, `unlink-4KB` and `copy-4KB`, respectively. Compared to EXT4, F2FS-F2FSJ demonstrates similar throughput to EXT4 in `create-4KB` and `unlink-4KB`. However, it outperforms EXT4 in `copy-4KB`, benefiting from F2FS's inherent advantage in sequential read/write performance. Conversely, F2FS-F2FSJ consistently outperforms XFS across all tested scenarios, achieving 1.21x, 2.7x, and 1.40x higher throughput on `create-4KB`, `unlink-4KB` and `copy-4KB`, respectively. This advantage stems from F2FSJ's decentralized journal architecture and its decoupled control and data planes.

Data-intensive workloads. As shown in Figure 10(c), for `Seq_write` and `Rand_write`, F2FS-F2FSJ drives IO bandwidth comparable with F2FS-CKPT. The reasons are that F2FS-F2FSJ incurs additional journal I/Os during data flush, and large file writes do not frequently trigger F2FS checkpoints. For `Seq_read` and `Ran_read`, F2FS-F2FSJ and F2FS-CKPT have almost the same bandwidth because large file read operations only modify one inode, resulting in negligible overhead for both journaling and checkpoint processes. Due to the appending nature, F2FS-F2FSJ and F2FS-CKPT have higher throughput than EXT4 and XFS on `Seq_write`, `Rand_write` and `Seq_read`. For `Ran_read`, all four filesystems exhibit the same bandwidth performance. Then, for mixed workloads in Figure 10(d), all four filesystems demonstrate similar bandwidth performance on `SeqWR_W` and `SeqWR_R`. Conversely, for `RanWR_W` and `RanWR_R`, F2FS-F2FSJ outperforms F2FS-CKPT, while EXT4 attains higher bandwidth than F2FS-F2FSJ due to interference on NAT in F2FS.

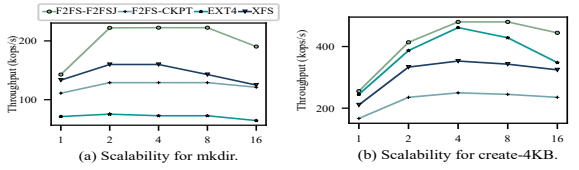


Figure 12: Scalability comparison for (a) mkdir and (b) create-4KB.

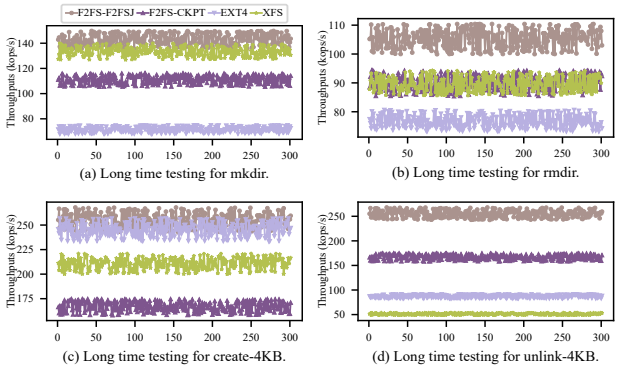


Figure 13: Long-term testing for (a) mkdir, (b) rmdir, (c) create-4KB and (d) unlink-4KB.

5.3.2 Results with Realistic Applications

Filebench workloads. Figures 11(a) and 11(b) shows the results with the filebench workloads. First, `OLTP` is a data-intensive workload and frequently invokes `fdatasync`, resulting in many I/Os to flush data. F2FS-F2FSJ has comparable throughput with F2FS-CKPT because checkpointing is not triggered frequently. Second, `Fileserver` is a small-file intensive workload, for which F2FS-F2FSJ shows 1.27x, 1.06x and 1.03x higher throughput than F2FS-CKPT, EXT4 and XFS, respectively. Then, `Webserver` primarily involves sequential read and write operations, for which F2FS-F2FSJ achieves 1.16x higher throughput compared to F2FS-CKPT, while both F2FS-F2FSJ and F2FS-CKPT demonstrate higher throughput than EXT4 and XFS due to F2FS's log structured design. For `Webproxy`,which is characterized by a high volume of file deletion and creation operations across 100 threads, F2FS-F2FSJ achieves 1.10x higher throughput compared to F2FS-CKPT but has lower performance compared to EXT4 and XFS. This is due to significant locking overhead in the filesystem metadata of F2FS, particularly in NAT and SIT, which leads to reduced performance. Finally, `Varmail` calls `fsync` every time after writes, incurring many I/Os to storage. For F2FS-F2FSJ and F2FS-CKPT, their performances are adversely affected by the frequent occurrence of I/Os during data flushing. EXT4 and XFS exhibits the lowest throughput due to the time overhead of journal period transfer.

Real-world applications. Figure 11(c) presents the results from real applications, including `git`, `cp` and `make`. First, for `git clone`, F2FS-F2FSJ incurs the lowest time cost and is approximately 5%, 18.3% and 21% faster than F2FS-CKPT, EXT4 and XFS, respectively. Second, for `cp -r` and `make linux`, which involve numerous reads and the creation of small files and directories, F2FS-F2FSJ achieves shortest

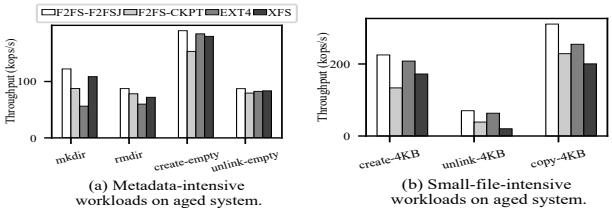


Figure 14: Throughput comparison on aged system for (a) Metadata-intensive workloads and (b) Small-file-intensive workloads.

execution times compared to F2FS (14.0% and 22.1% reductions), EXT4 (18.8% and 23.% reductions), and XFS (22.2% and 25.0% reductions).

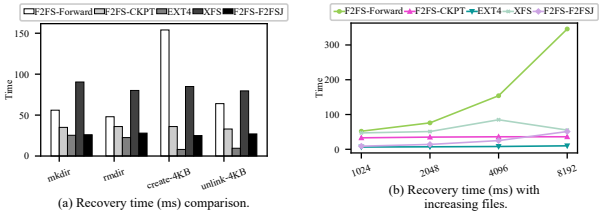
5.3.3 Scalability

We use `mkdir` and `create-4KB` to evaluate the scalability of F2FS-F2FSJ, F2FS-CKPT, EXT4 and XFS by setting the thread number from 1 to 16. Figures 12(a) and 12(b) show the results. For `mkdir`, F2FS-F2FSJ has the best scalability and throughput (e.g., 1.72x, 2.96x and 1.38x higher than F2FS-CKPT, EXT4 and XFS with 2 threads). Moreover, the performance of all filesystems begins to degrade when the number of threads exceeds eight [32, 46, 49]. For `create-4KB`, F2FS-F2FSJ has the best scalability and throughput (e.g., 1.92x, 1.03x and 1.35x higher than F2FS-CKPT, EXT4 and XFS with 4 threads). Conversely, F2FS-CKPT has the worst scalability because of its time-consuming and single-threaded checkpointing process.

5.3.4 Long-term and Aged Filesystem Testing

For long-term testing, we use four benchmarks: `mkdir`, `rmdir`, `create-4KB`, and `unlink-4KB`. We extend the original benchmark runtime by a factor of 300 (exceeding six hours for each long-term test). As shown in 13(a)–(d), the throughput performance of each filesystem remains stable, with fluctuations occurring within a consistent range throughout the long-term evaluation. Overall, F2FS-F2FSJ sustains the best performance without any sudden drops.

Aged filesystem testing is conducted using the Geriatrix [34] tool to simulate real-world fragmentation by distributing files of varying sizes across the filesystem. To evaluate performance under these conditions, we employ metadata-intensive and small-file-intensive workloads. As shown in Figures 14(a) and (b), F2FS-F2FSJ consistently outperforms F2FS-CKPT in an aged system environment, achieving 1.1x to 1.7x higher throughput. Furthermore, F2FS-F2FSJ demonstrates superior performance compared to EXT4 and XFS, as fragmentation significantly increases I/O overhead in JBD2 and XFS logging. Specifically, fragmentation causes excessive inode dispersion on the disk, leading to a rise in the number of bitmap pages requiring journaling. In contrast, F2FSJ's fine-grained journaling, which is based on metadata changes, remains unaffected by fragmentation.



(a) Recovery time (ms) comparison.

(b) Recovery time (ms) with increasing files.

Figure 15: Recovery time comparison for mkdir, rmdir, create-4KB and unlink-4KB: Recovery times for (a) creating or deleting 4,096 directories/files and (b) varying numbers of files.

5.4 Crash Recovery

In this section, we assess the crash recovery performance of various file systems and mechanisms, including F2FS-F2FSJ (F2FS with F2FSJ), EXT4 (EXT4 with JBD2), and XFS (XFS with its logging). Additionally, we evaluate F2FS-CKPT (F2FS with its default checkpointing and rollback recovery mechanism) and F2FS-Forward (F2FS with its roll-forward recovery mechanism).

Recovery time comparison. We utilize four benchmarks: `mkdir`, `rmdir`, `create-4KB` and `unlink-4KB`. Specifically, we create or delete 4,096 directories or files, then execute a checkpoint/journal commit, followed by a power failure. As shown in Figure 15(a), F2FS-F2FSJ reduces recovery time compared to F2FS-CKPT and F2FS-Forward. Under the `create-4KB` benchmark, F2FS-Forward has an unusually long recovery time. This is because it compares inodes before and after tagging to restore the data index, generating random I/Os that degrade performance. EXT4 has the shortest recovery time, as it writes back journaled metadata pages directly without reading the original metadata from the disk. In contrast, XFS has the longest recovery time because it replays the log twice - first to identify canceled items and then to process the remaining log entries while skipping the canceled ones.

Figure 15(b) shows the recovery time comparison across file numbers ranging from 1K to 8K. As observed, F2FS-CKPT maintains a stable recovery time of approximately 35ms, as it only restores a limited amount of filesystem metadata recorded in the checkpoint pack. Compared to F2FS-CKPT, F2FS-F2FSJ recovers faster when the file count is below 4K but experiences a 1.4x increase when handling 8K files. Additionally, F2FS-F2FSJ journal recovery significantly reduces recovery time - by 5.4x to 6.8x - compared to F2FS roll-forward recovery. Among all tested systems, EXT4 exhibits the shortest recovery time, as it only writes back journaled metadata pages without reading additional disk data. Meanwhile, XFS has a longer recovery time than both F2FS-F2FSJ and EXT4. Interestingly, when the file count reaches 8K, XFS recovery time decreases due to its proactive mechanism to apply certain log entries.

Crash consistency. We use CrashMonkey [43] to validate the correctness of F2FSJ recovery. Specifically, we evaluate rename workloads (`renam_root_to_sub`) and create/delete workloads (`create_delete`). The recovery results confirm that F2FS-F2FSJ successfully passes all test cases, demonstrating its consistency performance.

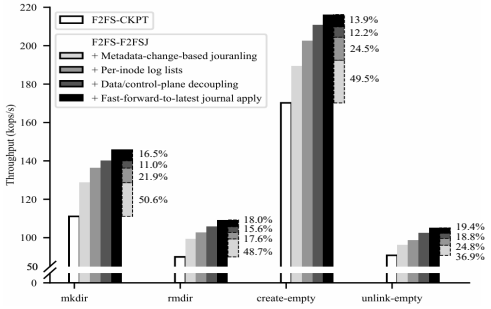
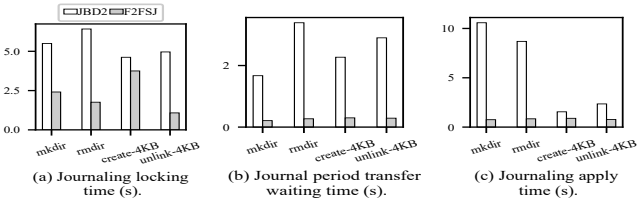


Figure 16: Throughput breakdown of each F2FSJ component for Metadata-intensive workloads.



(a) Journaling locking time (s).

(b) Journal period transfer waiting time (s).

(c) Journaling apply time (s).

Figure 17: JBD2 and F2FSJ journaling time breakdown: (a) Journal locking time. (b) Journal period transfer waiting time. (c) Journal apply time.

5.5 Breakdown Analysis of F2FSJ

In this section, we present a breakdown analysis of F2FSJ. First, we examine the contributions of its four components to the filesystem. Next, we compare the journaling time breakdown of F2FSJ with JBD2. Finally, we assess the impact of varying journal periods.

Effects of individual F2FSJ components. We analyze the throughput breakdown of each F2FSJ component using metadata-intensive workloads to assess their contributions. Figure 16 presents the throughput results of F2FS-CKPT (F2FS with default checkpointing) and F2FS-FSFSJ (F2FS with F2FSJ), highlighting the impact of each F2FSJ component: metadata-change-based journaling, per-inode log lists, data/control-plane decoupling, and fast-forward-to-latest journal apply. Among the total performance improvement, on average, metadata-change-based journaling contributes the most (approximately 49%) by leveraging fine-grained journaling to delay journal application, thereby reducing file operation blocking time. Per-inode log lists effectively minimize locking overhead during log collection, leading to a performance gain of approximately 21%. Data/control-plane decoupling reduces journal period transfer waiting time, resulting in a performance improvement of about 14%. Lastly, fast-forward-to-latest journal apply eliminates the overhead of across-epoch metadata application, contributing approximately 16% to overall system performance enhancement.

Journaling time breakdown. First, as shown in Figure 17(a), the locking time of F2FSJ is significantly reduced compared to JBD2, reduced by 18.8% - 78.2%. Second, the waiting time for journal period transfers in F2FSJ is noticeably less than that of JBD2, reduced by 86.8% - 92.0% in Fig-

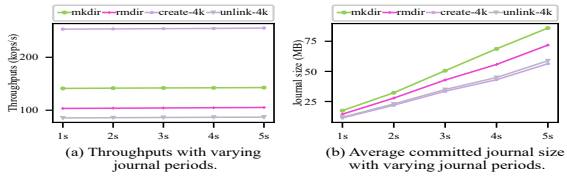


Figure 18: Throughput and journal size with varying journal periods.

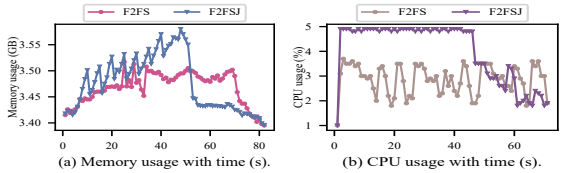


Figure 19: Overhead analysis: (a) Memory usage comparison. (b) CPU usage comparison.

ure 17(b). Third, in Figure 17(c), F2FSJ significantly reduces journal applying time (reduced by 68.4% - 95.3%).

Effects of varying journal periods. We evaluate the impact of varying journal periods on throughput and journal storage consumption. As shown in Figure 18(a), throughput across different benchmarks shows an increase (1.1%, 1.8%, 0.08%, and 1.6% for each benchmark) as journal periods extend from 1 second to 5 seconds, demonstrating that our decoupled control/data plane journal design imposes minimal overhead. In Figure 18(b), the average log size per journal commit exhibits an approximately linear increase, with maximum log sizes recorded at 85.9MB for `mkdir`, 71.8MB for `rmdir`, 56.4MB for `create-4KB`, and 58.8MB for `unlink-4KB`.

5.6 Overheads Analysis

In this section, we analyze the overheads associated with F2FSJ, specifically its impact on CPU usage, memory consumption, and storage utilization.

To assess CPU usage and memory consumption, we compare F2FSJ with F2FS checkpointing using `create_4KB`, which requires the highest memory and CPU resources. As illustrated in Figure 19(a), the memory usage of F2FS checkpointing ranges from 3.45GB to 3.50GB during the test, whereas F2FSJ exhibits a higher range of 3.50GB to 3.55GB, reflecting a modest 1.4% increase compared to F2FS checkpointing. For CPU usage, as shown in Figure 19(b), F2FS checkpointing averages around 3%, while F2FSJ reaches approximately 4.8% during the test. Consequently, F2FSJ exhibits 1.8% increase in CPU usage compared to F2FS checkpointing, which remains negligible.

For storage utilization, we compare F2FSJ (F2FSJ with F2FS) and JBD2 (JBD2 with EXT4), as both employ journaling mechanisms. In all the experiments discussed above, F2FSJ allocates 256MB for its journal file, representing only 0.1% of the total 256GB SSD capacity. In contrast, JBD2 requires 1GB of storage due to its page-based journaling approach. By leveraging metadata-change-based journaling, F2FSJ reduces storage usage to 25% of that required by JBD2.

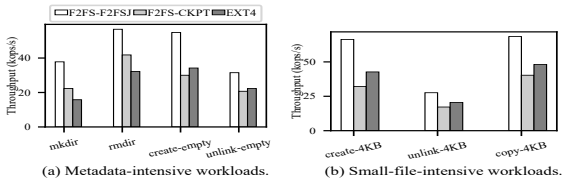


Figure 20: Throughput comparison for (a) Metadata-intensive-workloads and (b) Small-file-intensive workloads on ARM embedded board.

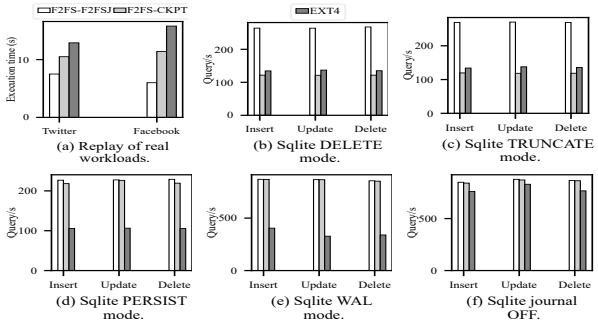


Figure 21: Real application experiments with Mobibench on ARM embedded board: (a) Execution time comparison for Twitter and Facebook workloads. Sqlite performance comparison for (b) DELETE mode, (c) TRUNCATE mode, (d) PERSIST mode, (e) WAL mode and (f) Journal-off.

5.7 Evaluation on ARM Embedded Board

For a more comprehensive evaluation, we expand our experiments to include micro-benchmarks and real-world Android applications on an ARM embedded board (Rockchip RK3588S [23]). This board features 4 Cortex-A76 and 4 Cortex-A55 CPU cores (8nm), 16GB LPDDR4 memory, and 128GB eMMC storage [6], running Linux kernel v5.10 and Android version 12 [2]. Specifically, we first compare the end-to-end throughput performance of F2FS-F2FSJ (F2FS with F2FSJ), F2FS-CKPT (F2FS with checkpointing), and EXT4 (EXT4 with JBD2) using micro-benchmarks. We then present results from Android applications, including execution time comparisons for Twitter and Facebook, as well as throughput comparisons for SQLite across different modes. Our evaluation shows that F2FSJ delivers superior performance improvements in micro-benchmark experiments compared to those observed in the desktop environment.

Metadata-intensive workloads. As shown in Figure 20(a), for metadata-intensive workloads, F2FS-F2FSJ significantly outperforms F2FS-CKPT by 1.70x, 1.36x, 1.83x and 1.52x and EXT4 by 2.39x, 1.76x, 1.60x and 1.41x for `mkdir`, `rmdir`, `create-empty` and `unlink-empty`, respectively.

Small-file-intensive workloads. As illustrated in Figure 20(b), F2FS-F2FSJ demonstrates significant throughput improvements compared to F2FS-CKPT, with gains of 2.06×, 1.60×, and 1.69× for create-4KB, unlink-4KB, and copy-4KB, respectively. Similarly, it outperforms EXT4 by 1.55×, 1.35×, and 1.42× for the same benchmarks.

Android workloads. First, F2FS-F2FSJ exhibits minimal time overhead for both Twitter and Facebook workloads, achieving time reductions of 28.5% and 47.3% compared to F2FS-CKPT, and 41.9% and 62.0% compared to EXT4, as

shown in Figure 21(a). Next, we evaluate SQLite throughput performance in terms of queries per second across different journal modes. Specifically, in Figures 21(b) and 21(c), F2FS-F2FSJ outperforms F2FS-CKPT by up to 2.24× and EXT4 by up to 1.96× for Insert, Update, and Delete workloads under both the DELETE and TRUNCATE modes. For the PERSIST and WAL modes, F2FS-F2FSJ demonstrates comparable performance to F2FS-CKPT while achieving up to 2.17× improvement over EXT4, as depicted in Figures 21(d) and (e). Finally, in Figure 21(f), F2FS-F2FSJ and F2FS-CKPT exhibit similar performance when SQLite journaling is disabled.

6 Related Work

JBD2. JBD2 utilizes a centralized approach with data/control coupling, where each epoch is associated with a global ticket and a lock. During file operations' journaling initiation/completion, acquiring the ticket lock to increment/decrement the ticket number is essential. Consequently, waiting times are inevitable during journal period transfers as the ticket lock must be obtained and the ticket number must become zero. In contrast, F2FSJ journals metadata changes in per-inode log lists, registering only the inode number in the epoch. Thus, it can issue the next epoch immediately during a journal period transfer without waiting for a global ticket.

NOVA. The per-inode log mechanism of NOVA [56] is tailored for NVM, which directly writes logs to the per-inode log area on NVM. In contrast, F2FSJ employs per-inode log lists for journaling, requiring different data organization and management mechanisms for log-structured filesystems.

WineFS. WineFS [33] is optimized for NVM. With undo journaling, it incurs more overheads for write transactions (i.e., record both old metadata/data); however, upon transaction completion, the associated logs can be discarded, eliminating the need for journal apply. This approach is effective for NVM, but slower I/O in flash storage may introduce more write overheads. In contrast, F2FSJ only journals metadata changes, enhancing journal write performance. Additionally, journal apply is not in the critical paths of data reads/writes, allowing it to be scheduled with minimal system impact.

BtrFS. Btrfs [50] optimizes updates to the copy-on-write trees [5] by journaling modified data and metadata related to a specific file in a dedicated log-tree. In contrast, in F2FSJ, we aim to design a novel journaling mechanism so as to provide fine-grained crash recovery for F2FS with different log organization and management schemes.

NV_LOG. NV_LOG [26] is a logging mechanism based on NVRAM (Non-Volatile Random Access Memory), which is not currently utilized in read-world products. In contrast, F2FSJ focuses on typical usage scenarios, offering a universal journaling approach tailored for flash storage devices.

I-journaling. I-journaling [48] enhances fsync performance through a hybrid journaling approach that combines JBD2 transactions with I-transactions. Per-CPU core I-transactions handle fsync operations, including the necessary metadata logs for crash recovery, while JBD2 transactions manage other file operations. This reduces fsync latency by committing I-transactions but still suffers from JBD2's locking overhead and extended journal transfer periods for non-fsync operations. In contrast, F2FSJ addresses these JBD2 limitations by employing a decentralized design and an epoch-based journaling mechanism, effectively eliminating locking overhead and journal transfer delays.

Z-journal. Z-journal [38] resolves JBD2's scalability challenges by introducing per-core journaling, allowing each CPU core to maintain its own transaction and using coherence commits for transactions that modify the same metadata pages. In contrast, F2FSJ separates journal data and control planes through per-inode log lists and epochs, achieving superior scalability and avoiding contention for journal operations on the same CPU core. Additionally, F2FSJ registers inodes into epochs, maintaining consistency without requiring complex coherence commits.

Fast-commit. Fast-commit [52] improves fsync performance and reduces I/O overhead in cloud environments by employing a hybrid journaling approach. JBD2 commits periodically (e.g., every 5 seconds), while Fast-commit logically journals filesystem updates within these intervals. It centralizes inode and dentry lists for journaling and uses compact logging to minimize journal size. However, as JBD2 continues running in the background, Fast-commit cannot fully eliminate JBD2's overhead and primarily optimizes fsync performance. In contrast, F2FSJ mitigates I/O overhead by journaling only metadata changes. Its decoupled data/control plane design enables fast journal period transfers, offering a more efficient solution than Fast-commit's hybrid approach.

7 Conclusion

In this paper, we propose a novel journaling technique, called F2FSJ, for F2FS with ordered journal mode. F2FSJ eliminates the long tail latency of checkpoint and provides fine grained crash recovery. F2FSJ incorporates several innovative designs. First, it only journals the metadata changes to reduce I/O and storage overheads. Second, it employs a decentralized journal design featuring per-inode log lists to minimize lock contention and interference. Third, it advocates an epoch-based strategy with decoupled data/control planes to eradicate waiting times during journal period transfer. Finally, it uses fast-forward-to-latest journal apply to consolidate multiple small writes. Evaluation results indicate that F2FSJ outperforms F2FS checkpointing.

Acknowledgments

We thank our shepherd, Kiran-Kumar Muniswamy-Reddy, and the anonymous reviewers for their constructive feedback and insightful comments. The work described in this paper is supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14202123, GRF 14200224).

References

[1] Android devices shut down in freezing weather. https://www.the-sun.com/tech/10094443/android-shutdown-keep-running-battery-cold-snow-weather.

[2] Android system wiki. https://en.wikipedia.org/wiki/Android_(operating_system).

[3] Bpf compiler collection. https://github.com/iovisor/bcc.

[4] Code comments of three scenarios cannot be roll-forwarded recovered of fsync in f2fs/recovery.c. https://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs.git/.

[5] Copy-on-write wiki. https://en.wikipedia.org/wiki/Copy-on-write.

[6] Emmc wiki. https://en.wikipedia.org/wiki/MultiMediaCard#eMMC.

[7] Ext4 wiki. ext4 disk layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[8] F2FSJ source code. https://github.com/10033908/F2FS-J.

[9] Filebench website. https://github.com/filebench/filebench.

[10] Filesystems supportted by android. https://source.android.com/docs/core/architecture/android-kernel-file-system-support.

[11] Fio documents. https://fio.readthedocs.io/en/latest/fio_doc.html.

[12] Flame graph. https://www.brendangregg.com/flamegraphs.html.

[13] Freezing weather can harm your phone. https://metro.co.uk/2024/01/17/iphone-android-battery-warning-issued-uk-users-20127874.

[14] fsync in linux man page. https://man7.org/linux/man-pages/man2/fsync.2.html.

[15] Fsync introduction. https://pubs.opengroup.org/onlinepubs/009695399/functions/fsync.html.

[16] Introduction of f2fs fsync mode. https://www.kernel.org/doc/Documentation/filesystems/f2fs.txtl.

[17] Introduction of free command. https://man7.org/linux/man-pages/man1/free.1.html.

[18] Introduction of poweroff command. https://man7.org/linux/man-pages/man8/halt.8.html.

[19] Introduction of top command. https://man7.org/linux/man-pages/man1/top.1.html.

[20] Linux source code. https://www.kernel.org/.

[21] Mobibench. https://github.com/ESOS-Lab/Mobibench.

[22] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[23] Product specification document of rk3588s. https://www.t-firefly.com/product/industry/rocrk3588spc.

[24] Three journal modes for jbd2. https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html.

[25] What are acid transaction? https://www.databricks.com/glossary/acid-transactions.

[26] Write anywhere file layout. https://en.wikipedia.org/wiki/Write_Anywhere_File_Layout.

[27] ALCORN, P. . Samsung releases new 12 gb/s sas, m.2aic and 2.5 nvme ssds: 1 million iops, up to 15.63 tb. http://www.tomsitpro.com/articles/samsungsm953-pm1725-pm1633-pm1633a,1-2805.html, 2013.

[28] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block io: introducing multi-queue ssd access on multi-core systems. In Proceedings of the 6th international systems and storage conference (2013), pp. 1–10.

[29] CHEN, F., HOU, B., AND LEE, R. Internal parallelism of flash memory-based solid-state drives. ACM Transactions on Storage (TOS) 12, 3 (2016), 1–39.

[30] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In 2011 IEEE 17th International Symposium on High Performance Computer Architecture (2011), IEEE, pp. 266–277.

[31] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In FAST (2012), vol. 12, pp. 101–116.

[32] EQBAL, R. ScaleFS: A multicore-scalable file system. PhD thesis, Massachusetts Institute of Technology, 2014.

[33] KADEKODI, R., KADEKODI, S., PONNAPALLI, S., SHIRWADKAR, H., GANGER, G. R., KOLLI, A., AND CHIDAMBARAM, V. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (2021), pp. 804–818.

[34] KADEKODI, S., NAGARAJAN, V., AND GANGER, G. R. Geriatrix: Aging what you see and what you {don't} see. a file system aging approach for modern storage systems. In 2018 USENIX Annual Technical Conference (USENIX ATC 18) (2018), pp. 691–704.

[35] KIM, D., MIN, K., OH, J., AND WON, Y. {ScaleXFS}: Getting scalability of {XFS} back on the ring. In 20th USENIX Conference on File and Storage Technologies (FAST 22) (2022), pp. 329–344.

[36] KIM, J. f2fs: introduce flash-friendly file system. https://lwn.net/Articles/518718/.

[37] KIM, J., CAMPES, C., HWANG, J.-Y., JEONG, J., AND SEO, E. {Z-Journal}: Scalable {Per-Core} journaling. In 2021 USENIX Annual Technical Conference (USENIX ATC 21) (2021), pp. 893–906.

[38] KIM, J., CAMPES, C., HWANG, J.-Y., JEONG, J., AND SEO, E. {Z-Journal}: Scalable {Per-Core} journaling. In 2021 USENIX Annual Technical Conference (USENIX ATC 21) (2021), pp. 893–906.

[39] LEE, C., SIM, D., HWANG, J., AND CHO, S. {F2FS}: A new file system for flash storage. In 13th USENIX Conference on File and Storage Technologies (FAST 15) (2015), pp. 273–286.

[40] LEE, E., SON, I., AND KIM, J.-S. An efficient order-preserving recovery for f2fs with zns ssd. In Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (2023), pp. 116–122.

[41] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of linux file system evolution. ACM Transactions on Storage (TOS) 10, 1 (2014), 1–32.

[42] MAO, B., WU, S., AND DUAN, L. Improving the ssd performance by exploiting request characteristics and internal parallelism. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37, 2 (2017), 472–484.

[43] MARTINEZ, A., AND CHIDAMBARAM, V. {CrashMonkey}: A framework to automatically test {File-System} crash consistency. In 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17) (2017).

[44] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In Proceedings of the Linux symposium (2007), vol. 2, Citeseer, pp. 21–33.

[45] MATHUR, A., CAO, M., AND DILGER, A. ext4: the next generation of the ext3 file system. Usenix Association 32, 3 (2007), 25–30.

[46] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding manycore scalability of file systems. In 2016 USENIX Annual Technical Conference (USENIX ATC 16) (2016), pp. 71–85.

[47] OH, J., YOO, S. W., NAM, H., MIN, C., AND WON, Y. {CJFS}: Concurrent journaling for better scalability. In 21st USENIX Conference on File and Storage Technologies (FAST 23) (2023), pp. 167–182.

[48] PARK, D., AND SHIN, D. {iJournaling}:{Fine-Grained} journaling for improving the latency of fsync system call. In 2017 USENIX Annual Technical Conference (USENIX ATC 17) (2017), pp. 787–798.

[49] PARK, J., HWANG, T., CHOI, J., MIN, C., AND WON, Y. {LODIC}: Logical distributed counting for scalable file access. In 2021 USENIX Annual Technical Conference (USENIX ATC 21) (2021), pp. 907–921.

[50] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. ACM Transactions on Storage (TOS) 9, 3 (2013), 1–32.

[51] SELTZER, M. I., GANGER, G. R., MCKUSIC, M. K., SMITH, K. A., SOULES, C. A., AND STEIN, C. A. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In 2000 USENIX Annual Technical Conference (USENIX ATC 00) (2000).

[52] SHIRWADKAR, H., KADEKODI, S., AND TSO, T. {FastCommit}: resource-efficient, performant and cost-effective file system journaling. In 2024 USENIX Annual Technical Conference (USENIX ATC 24) (2024), pp. 157–171.

[53] SON, Y., KIM, S., YEOM, H. Y., AND HAN, H. {High-Performance} transaction processing in journaling file systems. In 16th USENIX Conference on File and Storage Technologies (FAST 18) (2018), pp. 227–240.

[54] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In USENIX Annual Technical Conference (1996), vol. 15.

[55] WON, Y., JUNG, J., CHOI, G., OH, J., SON, S., HWANG, J., AND CHO, S. {Barrier-Enabled}{IO} stack for flash storage. In 16th USENIX Conference on File and Storage Technologies (FAST 18) (2018), pp. 211–226.

[56] XU, J., AND SWANSON, S. {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main

memories. In 14th USENIX Conference on File and Storage Technologies (FAST 16) (2016), pp. 323–338.

[57] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of nvme ssds and their implication on real world databases. In Proceedings of the 8th ACM International Systems and Storage Conference (2015), pp. 1–11.

Okapi: Decoupling Data Striping and Redundancy Grouping in Cluster File Systems

Sanjith Athlur[†*], Timothy Kim[†*], Saurabh Kadekodi[§], Francisco Maturana[†],
Xavier Ramos[†], Arif Merchant[§], K. V. Rashmi[†], Gregory R. Ganger[†]
[†]Carnegie Mellon University [§]Google

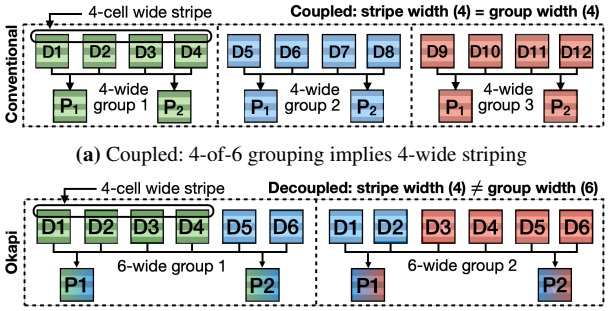
Abstract

The Okapi cluster file system decouples how data is spread across disks (data striping) for IO efficiency from how data is erasure coded together (redundancy grouping) for durability. Existing systems couple these two mechanisms' configurations, inducing significant inefficiencies. Decoupling allows grouping to be configured based on reliability and space efficiency goals, while simultaneously allowing striping to be configured based on performance goals. Decoupling also allows redundancy scheme changes from one EC scheme to another (e.g., to react to data temperature or disk failure rate changes) to occur without having to re-write data. Evaluation of an Okapi prototype shows that decoupling can be accomplished with <1% increase in metadata size and file manager memory, and minimal file creation and degraded read resource increase. Experiments demonstrate that decoupling can improve read throughput by 80% and reduce seeks per second by up to 70%, without yielding any data reliability, and reduce the overhead of redundancy transitions by up to 70%.

1 Introduction

Cluster file systems store data of a single file across multiple disks for IO efficiency and use redundancy to provide fault tolerance [7, 16, 50]. Although many environments use SSD caches and/or data replication for hot data, most data is stored via erasure codes on mechanical disks at lower cost-per-byte [4, 5, 48]. Generally speaking, k-of-n erasure codes (ECs) compute $r = (n-k)$ parity blocks over k blocks of data, and then store the n blocks on distinct disks. Such codes protect data from r failures at a space overhead of $\frac{n}{k}$.

Most existing systems spread each file's data across k data blocks, in units called *cells*, in a round-robin manner akin to RAID-4 [17]. This is called "striping". The size of a data block is typically 8MB or higher, and the size of a cell is typically 1MB. A file may consist of one or more of these arrays of k data blocks. Such striping improves degraded-mode read performance, decreases tail latencies, reduces buffer sizes for parity computation, and reduces capacity overhead for smaller files ($< k \times 8$ MB). The above described cell-level striping is used in Google's Colossus [25], Lustre [47], Ceph [11], HDFS [14] and many others [22, 39, 46, 57].



(a) Coupled: 4-of-6 grouping implies 4-wide striping

(b) Decoupled: 6-of-8 grouping while choosing 4-wide striping

Figure 1: 1a shows a file in a coupled architecture with a 4-of-6 EC scheme and 4-wide striping. Data is striped across 4 data blocks in a round-robin fashion as seen by the alternating light and dark colors. Parities are computed for every 4 data blocks. 1b shows the same file in Okapi's decoupled architecture with a 6-of-8 EC scheme and 4-wide striping. Data is striped across 4 data blocks, but parities are computed for every 6 data blocks. See Figures 3 and 4 for detailed examples.

In existing systems, each file's EC configuration (i.e., the values of k and r) dictates two decisions: how the file's data is spread across storage servers (i.e., data *striping*) and which blocks are encoded into the same parities (i.e., redundancy *grouping*). Specifically, data is striped over k blocks (stripe width = k), and those same k blocks (group width = k) are encoded together to form parities. Fig. 1a shows a file with 4-of-6 scheme where each row of 4 cells from 4 data blocks D1–D4, D5–D8 and D9–D12 are encoded to form parity cells (of the same size as data cells) in the two corresponding parity blocks. Thus, striping and grouping are tightly *coupled*. However, forcing stripe width and group width to be equal creates two fundamental problems: (1) an often too-limited set of options in a complex trade-off space between performance, space efficiency, and data reliability; and (2) worst-case re-encoding IO overhead when new circumstances dictate EC (grouping) changes.

First, with today's coupling, a file's values for k, r significantly affect its IO performance, space efficiency, and data

[*]Equal contribution

reliability. For instance, wider *stripes* (e.g., 30–96 wide [23, 25, 28]) allow higher bandwidth for huge sequential IOs, but can result in higher tail latencies and IO time used (due to extra seeks) when accessing smaller file ranges (e.g., <100MB), exacerbating the ever-worsening IO-per-TB-stored bottleneck [29]. Simultaneously, wider *groups* reduce storage overhead compared to narrower alternatives, such as 6-of-9 [13] or 10-of-14 [43], but increase reconstruction IO and can reduce data reliability. Many applications desire the performance benefits of narrow striping and storage overhead of wide grouping (such as video streaming services), or vice-versa. At Google, we observe many files use group width $k > 50$ to achieve space efficiency but suffer consequent IO inefficiencies, and other files use narrow schemes to reduce reconstruction overheads but suffer lower sequential bandwidth.

Second, coupling implies that changing a file's EC config, and hence group width (say from k_1 to k_2), requires re-*striping* its data contents. That is, all data units must be read and rewritten (striped across k_2 blocks), in addition to writing the newly computed parities. This overhead grows increasingly important as grouping changes become more common to reduce space overhead as data cools [29, 54], to maintain data reliability as disks age [25–27], and to address black swan events. One recent example of the latter at Google was a sudden failure rate increase of one disk make/model that dictated an emergency reduction of group width, overwhelming some storage clusters with a massive IO burst.

Recent work reported that millions of files transition to increasingly wider erasure codes every hour in Google clusters [29]. It highlights the massive scale of resulting IO demands and proposes efficient approaches to avoid them. However, unfortunately the IO savings are unrealizable for many systems using data striping because of the need for re-striping.

This paper introduces the Okapi cluster file system, which *decouples* each file's data striping configuration from its redundancy grouping configuration, enabling independent tuning. For example, Fig. 1b shows each range (e.g., 32 MB) of a file's data being striped over 4 data blocks (D1–D4, D5–D8 and D9–D12) while parities are computed over 6 blocks. And, if circumstances dictate a redundancy group width change (e.g., from Fig. 1b's 6-of-8 to Fig. 1a's 4-of-6 to increase data protection), it can be done without changing the stripe width or re-striping the data.

Achieving this increased flexibility requires careful design to avoid metadata blowup and minimize file creation and degraded read overheads, all without introducing complexities that might hinder adoptability. First, Okapi curbs metadata blowup by creating redundancy groups with sequential blocks of the file, which allows it to *infer* group constituencies from the existing stripe mappings instead of maintaining two different metadata structures. Second, Okapi uses *partial parities* to bound the client memory required during file creation. Specifically, when sequentially writing a file, computing parities can require data from multiple stripes containing data

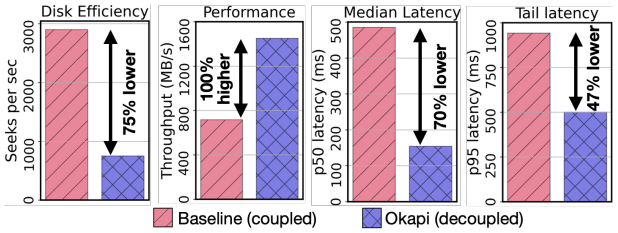


Figure 2: Decoupling improves cluster throughput, disk efficiency and workload latency. The graphs compare coupled and decoupled architectures for 8 MB reads on 12-of-15 EC files. While stripes are forced to be 12-wide in a coupled system, in Okapi, stripe widths could be optimally tuned independently of the EC scheme. See Sec 6 for detailed experimental setup.

blocks for a given group. Rather than caching all data until enough is written (which could easily be hundreds of MBs in wide groups), partial parities computed over accumulated data are kept instead. Third, Okapi smartly caches data during degraded-mode reads to limit read amplification.

We describe an implementation of Okapi in HDFS[1] to show how its concepts can be incorporated into an existing DFS that couples striping and grouping and to enable an apples-to-apples comparison to a coupled baseline. We note that other production scalable DFSs (e.g., Ceph, Panasas's PanFS and Google's Colossus) couple in a similar way, and could become decoupled with similar mechanisms. Experiments comparing to a coupled architecture with 6-of-9 EC scheme show that decoupling can increase read throughput by up to 80% (by reducing seeks by up to 70%) when matching stripe width to data access patterns, without compromising data reliability or space efficiency. With wider erasure codes such as 12-of-15 schemes, we observe up to 115% throughput improvement.

Fig. 2 shows a summary of performance benefits of tailoring stripe widths for 8 MB reads to 12-of-15 erasure-coded files. As a result of better IO efficiency, end-to-end client latency for a Google-derived read-only workload decreases by 36% when each file uses tailored stripe widths. Additionally, Okapi reduces IO for EC transitions by up to 50% compared to read–re-encode–write, and by up to 70% when used with other smart EC-transitioning techniques [29]. Analyzing a real emergency group width reduction scenario from Google and long-term disk-adaptive redundancy using failure logs from Backblaze production cluster confirm 38–45% less IO required for redundancy changes.

This paper makes five primary contributions. First, it exposes the unnecessary coupling of data striping and redundancy grouping in cluster file systems, analyzing resulting inefficiencies. Second, it provides insights into hyperscalar access patterns and workload trends through real-world application characteristics, and highlights the timely need to improve IO efficiency and reduce IO demand. Third, it introduces the decoupled approach, and identifies challenges in building a decoupled cluster file system—increased metadata, expensive file creation and degraded-mode access over-

[1] HDFS (Hadoop Distributed File System) is a popular open-source DFS, used in-production by more than 550 companies [10].

heads. Okapi overcomes them by (1) re-using existing structures and inferring EC group constituencies, (2) partial parity caching to curb client memory overhead, and (3) data caching to limit read amplification during degraded-mode reads. Fourth, it demonstrates that Okapi's ideas are easily adoptable in existing DFS by implementing them in HDFS. Fifth, via comprehensive evaluation on a real (academic) cluster using micro-benchmarks and Google-derived synthetic workloads, it shows that Okapi increases IO efficiency and reduces redundancy change costs.

2 Striping and grouping in cluster storage

Large cluster storage systems store exabytes of data on hundreds of thousands of mechanical disks (HDDs) housed in tens of thousands of servers. Data of a single file is divided into logically consecutive stripes. Each one of these stripes spreads its data in the cluster by *striping* the data across a number of disks and storing parities computed from that data *grouping* on other disks. This section serves to distinguish between striping and grouping in cluster file systems.

Striping: spreading data and accesses. Cluster storage systems break files into ranges and spread them among disks as discussed above. Indeed, most cluster storage systems go further, striping data among data blocks in a round-robin fashion within each range at a finer granularity (generally referred to as a *cell* or a *striping unit*) to improve performance without increasing metadata overheads. The number of data blocks a stripe spans across is called the *stripe width*. Fig. 3 shows a 64 MB file with a stripe width of 4 and a cell size of 1 MB. Note that 4 MBs can be read by fetching 1 MB from 4 disks in parallel.

The striping configuration (cell size and stripe width) dictates important aspects of disk efficiency and performance observed by clients. For every IO, a disk must position the read/write head (seek plus rotational latency) and then transfer the data to/from the disk surface. While both consume "disk time", a highly constrained and shared resource, only the data transfer part is useful. Thus, fewer larger transfers are more efficient, as they amortize positioning time over more useful work. Also, wider stripes can increase tail latency, since a client request is not complete until all involved disks are done. But larger transfers do take longer to complete, and thus can increase latency for clients when cluster load is low (i.e., when extra seeks wouldn't cause queueing delays). Generally, wider striping can be better at low load and worse at high loads, with the trade-off being most relevant for mid-sized requests. We define *mid-sized* requests as those that access more than one data cell but not so large that any choice will involve all disks containing relevant data blocks.

Grouping: erasure-code redundancy. Device failures are common in exascale cluster storage. Thus, data is stored redundantly to protect against permanent data loss. While certain hot data may be replicated, most data is erasure-coded

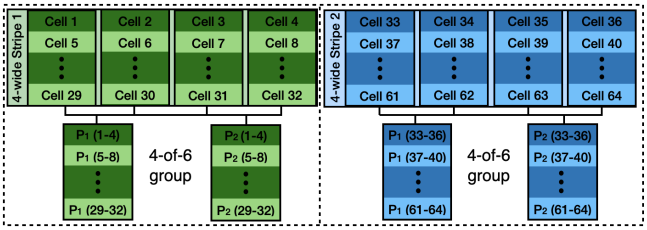


Figure 3: Coupled Architecture of a striped 64 MB *file* with 1 MB cells. Each column of cells is a data block, and data blocks of a stripe are stored on unique disks. The file consists of two stripes: blue and green. This coupled file is forced to use 4-wide striping due to using 4-of-6 grouping.

[13] with each group of data blocks encoded into parity blocks stored on other disks in the cluster (Fig. 3).

When using k-of-n Reed-Solomon erasure codes (EC), each of the k data blocks are encoded into $r = (n-k)$ parity blocks such that any k of the n blocks can provide all k data blocks. Naturally, when possible, a system will read the data blocks rather than using the parities to reconstruct.

The grouping configuration (k and n) dictates important aspects of data reliability, capacity efficiency, and reconstruction overheads. Parities add a capacity overhead of $\frac{r}{k}$. Unsurprisingly, more redundancy increases data reliability. In addition, reconstruction of missing data requires k of the n blocks of data. The reconstruction costs are therefore directly proportional to the value of k. In most environments, a lower bound of r (e.g., 3) is fixed due to factors such as datacenter makeup or planned maintenance outages [27, 28], while the group width (k) can vary. Widths like 6 [13] and 10 [43] and beyond [3] are now common, with 95 and 150 recently being reported [12, 28]. Data requiring higher reliability or being stored on more failure-prone disks is usually stored using *narrower* groups, accepting higher capacity overhead.

3 Why decouple striping from grouping?

Several factors related to IO efficiency determine the best stripe width for a given file, while independent factors related to reliability and space efficiency determine the best group width. Due to today's coupled striping and grouping, applications must choose among limited options (i.e., stripe width must equal group width) often inducing sub-optimal configurations for one or both widths. Furthermore, a coupled architecture forces all data to be re-written when changing EC schemes, inducing significant IO overheads. Emerging storage hardware trends, insights into datacenter workload access patterns, and modern redundancy management techniques further motivate the need for decoupling.

Storage hardware trends. HDDs are becoming denser without becoming proportionately faster [29]. Due to the decrease in IO-per-capacity, disk seeks and disk bandwidth are becoming an increasingly scarce resource. Thus, it becomes critical to improve both the IO efficiency of workloads (by tailoring stripe widths) and reduce IO from workloads that consume high disk bandwidth (e.g., EC transitions).

To demonstrate the impacts of coupling on IO efficiency, consider reading 12 MBs of data from a 6-of-9 erasure-coded file. Assuming 10 ms average positioning time (seek plus rotation) and sustained 150 MB/s data transfer rate, reading 2 MBs from 6 disks consumes 40% more aggregate disk time than reading 6 MBs from 2 disks[2]. To optimize for IO efficiency, if stripe width is set to 2, the file would need to use a 2-of-5 redundancy scheme (to be able to sustain 3 failures), resulting in an unacceptable 2.5× storage overhead.

Hyperscalar workload trends. As hyperscalars store more total data, the average temperature (popularity) of data in their storage clusters gets colder. The steady proportionate growth of cold data has promoted the adoption of *wide erasure codes* ($n \geq 20$) [3,28], which offers excellent capacity-efficiency but suffers from poor IO efficiency. This is especially true for *mid-sized* reads (>1 MB and ≤≈50 MB) where reading from wide EC incurs several 1-2 MB IOs, each one paying heavily in positioning time. At Google, we observe that such mid-sized requests contribute to 65% of total *bytes read* (see Fig. 6a for read-size distribution), which would immensely benefit if stripe widths are tailored to access patterns.

Moreover, we observe several real-world examples of this tension: (1) user-facing media streaming services want *narrow stripes* for IO efficiency and lower tail latency but *wide groups* for capacity efficiency; and (2) big-data processing workloads favor *wide stripes* to increase bandwidth for their large sequential reads but prefer *narrow groups* to reduce reconstruction costs upon failure. A coupled design locks in fundamental tensions between performance and capacity.

Redundancy adaptation trends. Recently proposed disk-adaptive redundancy systems [25–27] improve capacity efficiency by dynamically tailoring data redundancy to changing disk failure rates. Other systems [29] optimize transition IO overheads over a file's lifetime as it cools. However, these systems all fail to address the implicit degradation of IO efficiency due to these changes. For e.g., a 10 MB read to a 10-of-13 file after transition incurs a 2× increase in disk seeks v/s a 10 MB read to the same file in 5-of-8 before transition.

At Google, although the redundancy scheme of a file commonly changes throughout its stored life, the file's *access pattern usually remains constant*. We measured request-sizes of reads to uniformly sampled subset of files across three large Google storage clusters. Across these clusters, we found that 64%–94% of the sampled files were accessed using the *exact same read size* over a period of 150 days, whereas its EC scheme changed up to 4 times in the same duration. Furthermore, the per-file read size distribution exhibited very low variance with most reads heavily concentrated around the mean. This highlights the opportunity to tailor stripe widths of a file to improve throughput and resource efficiency. This result is unsurprising as most accesses in exascale cloud environments are issued by internal services or applications, which have known access patterns, and not direct user reads.

Increasing EC transitions trends. Transitioning data from one k-of-n scheme to another has become increasingly common. At Google, the most common transitions involve increasing group width for data as it cools. We observe over 100K EC-to-EC file transitions per day, resulting in multiple petabytes of disk IO every day in large clusters at Google.

Bulk redundancy transitions can also happen in response to datacenter emergencies. For example, in late 2020, one exascale cluster observed sudden HDD failures (>4× the usual rate), triggering an unusually high number of data reconstruction events. To mitigate the issue, the data was transitioned from $k \approx 50$, which has high reconstruction-IO costs, to a smaller k (≈15) which is more reconstruction-IO-efficient.

The above redundancy adaptive systems heavily rely on redundancy transitions for realizing their benefits. These scenarios have elevated redundancy transitions to a first-class operation akin to reads and writes. However, despite the many optimizations proposed for reducing transition IO in the above systems [26, 29, 35, 36, 59], they are rendered inconsequential when files are striped in a coupled architecture since re-encoding forces re-*striping*. Not only does this reorganization make transitions costlier, it also induces unnecessary IO efficiency problems (i.e., the new stripe width can be less IO-efficient for the workload) as stated above.

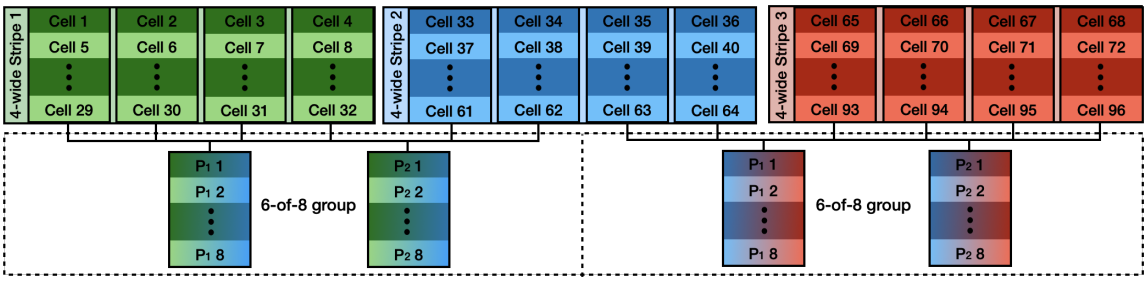
We now present Okapi, a decoupled system that gracefully tackles the aforementioned trends.

4 Okapi's Design for Decoupling

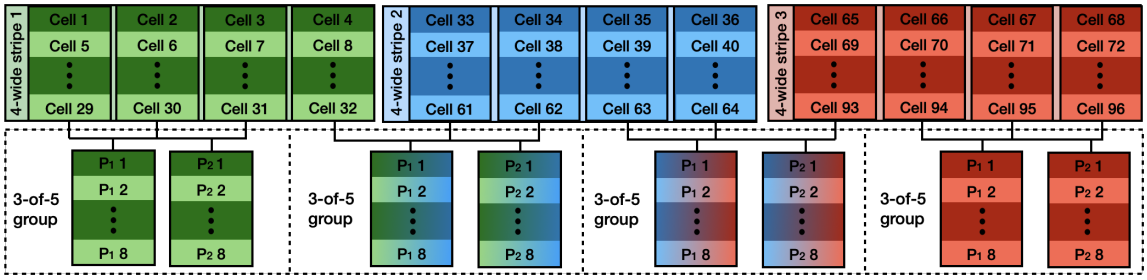
The Okapi cluster file system allows independent configuration of data striping and redundancy grouping for *each file*. Okapi stripes a file across blocks and encodes blocks into parities like existing cluster file systems, but does not restrict group width to equal stripe width. Each file's contents are stored in one or more fixed-size blocks. Unlike existing systems, applications specify stripe width, in addition to grouping configuration (k and r). Data is striped MB-by-MB over each consecutive set of *stripe-width* blocks in a round robin fashion, and r parities are computed over each *consecutive* k blocks of the *same* file, no matter how the data is striped. For performance and durability purposes, Okapi requires blocks in a stripe *and* blocks in a group to be stored on unique nodes/failure domains. As described later in this section, such a design allows for lower per-file metadata, and curbs file creation and degraded-mode read access overheads. Fig. 4 illustrates the data layout for 4-wide stripes, but with two different grouping schemes: 6-of-8 and 3-of-5.

Next we describe key challenges involved with decoupling and how they are addressed in Okapi.

[2] 6×10ms for positioning plus 80ms of data transfer vs. 2×10ms for positioning plus 80ms of data transfer. Note that, in each second, each disk has one second worth of time divided between positioning and data transfer (and potentially idle time), neither of which get much faster with newer disks.



(a) 96 MB file with 4-wide stripes and 6-wide groups.



(b) 96 MB file with 4-wide stripes and 3-wide groups.

Figure 4: Decoupled Architecture: A 96 MB *file* striped 4-wide but shown with two different EC schemes - (a) 6-wide and (b) 3-wide. Each color—green, blue and red, represents a different stripe. Each 4-wide stripe is composed of **4** data blocks storing 32 MB. A group is composed of consecutive data blocks of the file, that may cross stripe boundaries. Block x is mapped to stripe $\lceil \frac{x}{\text{stripe width}} \rceil$ and group $\lceil \frac{x}{\text{group width}} \rceil$.

4.1 Challenges: decoupling stripes and groups

Increased metadata. Conventionally, EC file metadata contains the redundancy scheme configuration and location information (list of the disks) for each stripe. In a decoupled design, the DFS must also track the location information for each EC group. This additional metadata (usually stored in main memory) must have a small footprint, as memory is a valuable resource in any large-scale storage cluster.

Expensive file creation/appends. In a decoupled design, multiple groups can be incomplete at any given time while writing a new file, as opposed to just one in conventional systems. All data of incomplete groups need to be retained in memory until the parities are computed and flushed to the disk. This results in an increased client-memory footprint. A decoupled design must minimize memory requirements, and reduce unnecessary buffering of large amounts of data.

Increased work during degraded mode reads. During degraded mode reads, it is necessary to fetch at least k out of the n chunks from the group, even if they are not directly requested. This read amplification can be exacerbated in some decoupled configurations, since contents of stripes accessed is different from contents of groups required for reconstructing missing data. Such read amplification due to decoupling must have minimal impact on cluster throughput and latency.

Adoption-friendly design changes. Large-scale DFS usually take upwards of ten years to fully mature and become widely adopted. Proposing a design that warrants a DFS to be built from ground-up is therefore non-viable. A decoupled system should introduce straightforward design changes such that they can be readily incorporated into existing DFS.

4.2 Reducing metadata by inferring groups

Traditionally, DFS maintain the list of stripes that make up a file, along with the set of nodes where the data and parity blocks of each stripe can be found. Since stripes and groups are distinct in Okapi, we now need to maintain location information for both components separately, doubling the per-file metadata, which may be prohibitive.

Since redundancy groups in Okapi are comprised of *consecutive* data blocks, Okapi simply *infers* the redundancy group mapping from the existing data stripe mapping based on the stripe width and the file's EC scheme. Imagine the data blocks of a file (across all stripes) sequentially numbered from 1 to X. Then, data block x is mapped to stripe $\lceil \frac{x}{\text{stripe width}} \rceil$ and group $\lceil \frac{x}{\text{group width}} \rceil$ within a file. Given the ordered sequence of data and parity blocks, irrespective of the corresponding stripe information, Okapi can easily map the i^{th} set of k data blocks to i^{th} set of r parity blocks using simple modulo operations to identify redundancy groups of the file. We show the marginal metadata overhead incurred from using this inference logic in a live cluster instance in Sec. 6.

4.3 Partial parities for efficient file append

A common client access pattern is to create an EC file, write sequentially, and never modify the file once it is closed. Indeed, this access pattern is so dominant that popular DFS (e.g. HDFS, Ceph, Microsoft Azure) do not support modify or append operations on EC files [11, 14, 24].

In most DFS, when a file is written in this pattern, the system will wait to compute parities until all k data cells

of the row of a group are available. Although the written cells remain unprotected until the kth cell is received and the parities are computed and written out, this approach is much easier and more efficient than stricter durability semantics. Okapi retains the same semantic to allow for fair comparison with a coupled system to focus on the effects of decoupling.

To achieve the expected efficiency, the client must keep a copy of yet-unprotected cells in memory. For instance, consider writing the group shown in Fig. 3. The first 3 cells are written out to disks but also buffered in client memory until the 4^{th} cell becomes available to compute the first parity cell $P_1 1(1-4)$. Using this approach, decoupling can increase the load on client memory in some configurations. For example, for 6-of-8 groups with 4-wide striping as shown in Fig. 4a, the client would need to buffer the first 34 cells in order to compute the first parities. Okapi mitigates this overhead by computing intermediate *partial parities*. Rather than waiting for all data in a redundancy group to be written, Okapi greedily computes parity blocks with currently available data, which it keeps in memory instead of the more numerous data cells. We describe Okapi's partial parity computation below.

For a k-of-n erasure code, parity blocks for a vector of k data blocks are computed by using a $(n-k) \times k$ generator matrix. This generator matrix ($G_{(n-k)\times k}$) is simply multiplied with the data vector ($D_{k\times 1}$) over a Galois Field to produce the parity vector ($P_{(n-k)\times 1}$). For a simple 3-of-5 EC scheme, the parities are computed as follows:

$$G_{2\times 3} \times \begin{bmatrix} D_0 & D_1 & D_2 \end{bmatrix}^T = \begin{bmatrix} P_0 & P_1 \end{bmatrix}^T$$

Okapi breaks this into k *independent partial* computations.

$$G_{2\times 3} \times \begin{bmatrix} D_0 & 0 & 0 \end{bmatrix}^T + G_{2\times 3} \times \begin{bmatrix} 0 & D_1 & 0 \end{bmatrix}^T + G_{2\times 3} \times \begin{bmatrix} 0 & 0 & D_2 \end{bmatrix}^T = \begin{bmatrix} P_0 & P_1 \end{bmatrix}^T$$

A partial parity block can be computed for every data block as soon as they become available without needing any of the other data blocks. For example, for D_0

$$G_{2\times 3} \times \begin{bmatrix} D_0 & 0 & 0 \end{bmatrix}^T = \begin{bmatrix} P'_0 & P'_1 \end{bmatrix}^T$$

Rather than buffering all the data, Okapi instead buffers partial parities in *client memory*. Computed partial parity blocks are then simply added, as they become available, to eventually form the final parities. Breaking up the computation of parities by exploiting the associativity of a linear operation is an optimization that has been used in several prior contexts [32, 37, 40, 60]. Okapi re-uses the concept in the write path to mitigate the overhead introduced by decoupling.

Note that only the final parities are flushed to disk. Storing partial parities in client memory does not change or compromise the durability guarantees in the case of a client failure. In a coupled DFS, a client failure triggers a background recovery process which checks the last stripe in the file and generates parities if the stripe is under-protected. Similarly in Okapi, the recovery process checks for any under-protected redundancy groups and generates new parities for each by reading from data blocks. The recovery process in Okapi makes no use of partial parities and maintains the same durability semantics.

Distributed file systems such as HDFS and Okapi can be modified to support sync-on-write semantic, but doing so would hurt write performance substantially. Client write latency would not only see the time to receive a response for data written out to servers but also the time to send new data to update existing parities. Furthermore, the file system would need to support atomic read-modify-writes to incrementally update parities, either via shadow-paging or versioning. The required functionality and associated overheads would be necessary in systems both with and without decoupling.

4.4 Caching to reduce degraded read cost

If some blocks are unavailable during a read (e.g. due to a failed disk that is yet to be recovered), they have to be reconstructed on the fly at the client. Such degraded reads need to access both the data stripe (for read) and redundancy group (for reconstruction).

In a normal read operation, the client only reads the required cells. For degraded reads, at least k of n cells from the group containing the missing data need to be fetched, which may not have been read by the client otherwise, causing read amplification. In certain scenarios, Okapi's decoupling can increase this read amplification. Consider a 6 MB read in 6-of-8 EC file shown in Fig. 4a. Cells 1-6 need to be accessed (across two stripe rows). If the first block is missing, Okapi will read 12 MB of total data for on-the-fly recovery. A coupled system, with 6-wide group, would only read 6 MB. However, such an increase in read amplification occurs rarely.

For reads requesting less than one stripe row worth of data (small-sized requests), the degraded mode behaviour remains unchanged. For larger reads, in particular, reads spanning multiple stripes of data or an entire file (large sequential requests), the data required for reconstruction will either have been recently read by the client or will be fetched by subsequent reads. Okapi exploits this by anticipating a degraded mode read and caching the required data as it is read to avoid fetching the same data multiple times, eliminating read amplification.

For other mid-sized requests, although degraded-read performance in decoupled cases can be worse, we find that the overhead with Okapi's design is minimal and is an acceptable trade-off. We quantify these overheads in Sec. 6.4.

4.5 Re-grouping to minimize EC transition IO

EC transitions are commonplace in large-scale storage clusters (Sec. 3). We describe how Okapi's decoupled design helps in significantly reducing the IO cost of transitioning.

Storage clusters require sequential data to be written over consecutive stripes to exploit maximum parallelism and to ensure maximum disk efficiency while reading any range of bytes of data. In a coupled system, this strict restriction gets unnecessarily imposed on redundancy groups too. As a result,

when transitioning a file from k_1-of-n_1 to k_2-of-n_2, conventional file systems are forced to re-*stripe* the file - read the entirety of the file's contents and write them back according to the new striping layout like a new file. For example, if the stripe width of the file needs to be changed from 4 to 6 in Fig. 3, almost all the cells need to be moved from their original disk locations to different disks, incurring significant disk IO and network overhead.

A decoupled system allows us to instead *re-group* the file and change the EC groups without modifying the striping layout. Okapi simply determines which data blocks will make up the new groups and recomputes new parities for them without moving any of the data blocks. All data blocks are read, but only parities (and not data) are re-written. Consecutive sets of k_2 *data* blocks that may be spread across multiple stripes of the file are chosen to form $C = LCM(k_1, k_2)/k_2$ new redundancy groups. The amount of data to read can be further reduced with smart EC transitioning techniques [29].

In order to ensure maximum durability, the physical blocks that make up an EC group must exist on disks in different failure domains. When such re-grouping is performed, new groups may end up with multiple physical blocks that reside on disks in the same failure domain. In such situations, Okapi simply relocates those blocks such that a group is composed of blocks on distinct disks and incurs the cost of this extra data movement. However, these overheads can be completely eliminated by being cognizant of the possibility of redundancy transitions and allocating data of successive groups of a file to distinct disks during its creation. Such mindful placement mechanisms should be straightforward since mechanisms already exist to form n-wide groups.

4.6 Choosing stripe widths and group widths

Most data in hyperscalars are created and used by large applications/services that have engineers dedicated to maximizing their efficiency. Such engineers measure their system in operation and analyze trade-offs to tune their coupled stripe/group width configuration to meet desired reliability targets, minimize byte overhead, and achieve needed throughput/latency and disk-time efficiency [53, 55, 58].

At Google, for instance, engineers for large-scale data services periodically choose a space-efficient (coupled) EC scheme for their data that meets their reliability and performance SLOs. Reliability SLO is described using MTTDL. MTTDL, governed by the group width, can be calculated either analytically (using Markov chains) or experimentally (using Monte-Carlo simulations) [49]. Meeting the performance SLO on the other hand requires careful benchmarking. This entails generating artificial data using the proposed EC scheme using the application's preferred read size distribution. Average and tail read latencies are then calculated for normal reads and degraded reads performed by injecting failures. These reads are performed at normal *and* high cluster load to ensure that system performance continues to remain stable.

Okapi can make such selection processes easier in that a single width choice does not have to satisfy all reliability and performance goals. Our experiments indicate that a stripe width of $\lceil \frac{\text{median read request size}}{\text{block size}} \rceil$ works well in optimizing throughput and median latency with various group widths (which can be chosen for reliability), by amortizing seeks over substantial data transfer (see Sec. 6.5). But, *consistent with current practice*, we expect most applications/services to use benchmarking in configuring their (now-decoupled) stripe and group widths. We extract salient features from the aforementioned Google EC scheme selection process into *a* generalized strategy:

1. **Choose an EC scheme k-of-n.** Balance target MTTDL, byte overhead, and reconstruction costs [17, 20, 26, 27].
2. **Record IO patterns.** Randomly sample f representative application files. For each, log read request sizes over a representative time window that captures typical usage patterns, such as a full day during steady-state operation. Many organizations already have well-established observability infrastructure [51] which can be readily leveraged to collect and analyze I/O read sizes.
3. For each candidate stripe width x:
   (a) **Generate test files.** Create and write f files with group width k and stripe width x.
   (b) **Replay IO logs.** Issue the recorded read requests against test files in the same live cluster environment to capture realistic performance. Measure per-request latency.
   (c) **Analyze performance.** Measure median, tail latencies.
4. Choose the group and stripe width that satisfies all SLOs and offers the best balance of read performance (throughput, median latency, and tail latency) for future files to be created. Additional metrics such as seek count and write throughput may also be recorded and taken into account. There may be no single configuration that is optimal across all metrics; application engineers make subjective decisions based on which performance aspects are most critical for their specific use case.

Since IO requests to real, existing application files are sampled to collect the IO logs, access patterns, read size distribution and file size distribution are all inherently captured in this benchmarking process. If it is infeasible to record IO access patterns, or if IO access patterns are known to frequently change over time (a rare occurrence, see Sec.3), applications can default to setting stripe width equal to group width, performing no worse than coupled systems.

5 Okapi's Implementation

Okapi is implemented by modifying the Hadoop Distributed File System (HDFS). We use existing data structures and tried-and-tested pipelines as a conscious choice to not hurt the current performance or integrity of existing mechanisms

in HDFS[3]. This section describes how Okapi efficiently decouples stripes and groups, and the mechanisms that enable file read/write, data protection, and EC transitions. We briefly describe how decoupling can be implemented in few other popular DFS in Appendix A.

5.1 Simple, efficient decoupling

Traditional HDFS maintains metadata for a stripe of data blocks and its corresponding set of parity blocks in a data structure called striped block groups (SBG), which is tracked as a single entity in the Namenode's BlockManager.

Metadata. Okapi splits a conventionally coupled SBG into two structures: a data stripe and a parity group. A data stripe is a logical ordering of data blocks and a parity group is a set of parity blocks. This split allows the structures to have a O(1) lookup time in the BlockManager as they were for SBG's.

To minimize metadata overheads, Okapi saves 8 bits per block by eliminating the pointer to the EC policy and leverages the previously unused replication field instead. Additionally, it ensures sequential ordering of storage information, removing the need for internal physical block ordering in the new data structures. To minimize inode metadata, Okapi uses a single byte (max value=127, easily extendable) to track the stripe width per file in the header. This approach supports most widely published schemes to date [28].

Writes. Okapi allows files to be written with any arbitrary stripe width and EC scheme. It uses partial parities to avoid buffering large amounts of data between stripes. The partial parities are stored in a r-wide array of byte buffers, each of which stores data worth 1 block. This guarantees that the client will buffer at any point at most block size $\cdot\, r +$ cell size $\cdot$ k of data. When updating partial parities, Okapi multiplies only the relevant subsection of the encoding matrix with the data vectors to eliminate extraneous computation. This aspect of the implementation ensures that encoding and combining partial parities is computationally equivalent to encoding full parities in a single operation.

Normal reads. Normal reads in Okapi work exactly like in HDFS. Okapi simply looks at data stripes to inform clients where to read data from, similar to how SBG's are used, Parity groups are not needed and remain unused. This ensures that normal read performance in Okapi is not impacted at all by decoupling and that no additional necessary computation is required to retrieve physical block locations.

Degraded reads. A degraded read will infer the redundancy grouping for the failed block to reconstruct missing data. Okapi's design prioritizes efficiency and attempts to re-use any data that was already buffered in from the read when decoding the corrupt or missing data.

Under-replication detection and recovery. Data stripes and parity groups are managed in the exact same way as SBG's. Okapi therefore re-uses existing block reporting system for under-replication detection, with no loss in performance. The BlockManager lazily computes the redundancy grouping once under-replication is identified using constant time look-ups and modulo arithmetic.

Efficient and safe EC transitions Okapi computes new grouping based on the file's array of data stripes and the new EC scheme. After the new groups are determined, the new parity groups are added to the BlocksMap and the new grouping is sent to the Datanodes using HDFS's existing EC recovery pipeline. The new parities are encoded and reported by the Datanodes, and the old parities are atomically removed from the Namenode. This ensures that files are protected even during a transition as the old parity groups are available until the new parity groups are completed.

6 Evaluation

This section evaluates Okapi and confirms three main takeaways: (1) configuring a file's stripe width to match data access pattern provides significant IO efficiency improvements without compromising reliability. (2) decoupling stripe width from group width significantly reduces the IO required for EC scheme transitions. (3) Okapi does not introduce significant metadata, file creation or degraded-read overheads.

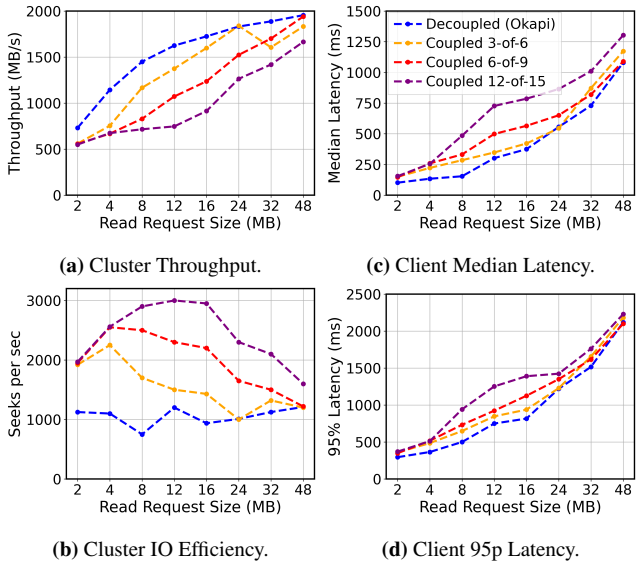
6.1 Experimental setup

Both Okapi and coupled experiments run on a 20-node cluster with one Namenode and 19 Datanodes. Each machine has a Quad-Core AMD Opteron Processor and 128 GB RAM. Each Datanode has a 1 TB Ext4 file system atop a 7200 RPM Hitachi Ultrastar HDD. The physical block size and file IO buffer size in HDFS are set to 8 MB (default block size at Google). The underlying Ext4 file system uses a 1 MB block size, which is the same as the cell size. Cluster nodes are connected via a 40GbE network. We measure throughput with DFS-perf [21]. For each workload, we measure aggregate throughput and median, p95, and p99 client request latency. We examine block device IO and disk seek behavior using blktrace [6] and seekwatcher [33]. We measure IO and CPU/memory costs by aggregating per-node data exposed by Ganglia [34]. We compare Okapi's performance with HDFS with micro-benchmarks and then with a synthetic trace generated using read-size distribution drawn from Google's cluster.

6.2 Decoupling improves IO performance

Our results demonstrate a significant benefit to total cluster read bandwidth and client latency when configuring the data striping to be tuned for their specific access patterns, without impacting factors such as storage overhead, reconstruction costs, which are controlled by grouping. We first compare read throughput and IO efficiency across varying access patterns

[3] Okapi is open-sourced at https://github.com/Thesys-lab/okapi.



(a) Cluster Throughput. **(c)** Client Median Latency.

(b) Cluster IO Efficiency. **(d)** Client 95p Latency.

Figure 5: Performance measurements for requests ranging from 2 MB–48 MB for files grouped 3-of-6, 6-of-9, and 12-of-15 (and thus striped 3-, 6-, and 12-wide respectively for HDFS) and pre-determined stripe width with 6-of-9 grouping for Okapi. Compared to 6-of-9 coupled configuration, throughput improves by up to 80% and seeks/sec reduce by up to 70% when decoupling with similar benefits to median and tail latency.

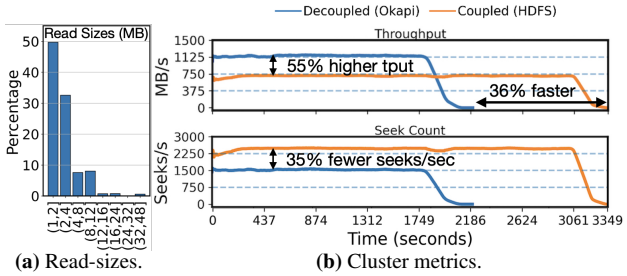
with/without decoupling. Then, we characterize the benefits of decoupling in a realistic read-only system workload. Finally, we show the impact of decoupling on write throughput.

Tuning stripe width improves read performance. We present cluster and client metrics of a read workload in which both HDFS and Okapi use 6-of-9 grouping (a popular EC scheme [13, 27]), which means that both systems have the same storage overhead. While this forces files in a HDFS to be striped 6-wide, Okapi faces no such limitation and allows clients to set the file's striping width based on its most common access pattern. In addition to 6-of-9 grouping, to evaluate alternate options, we include results for HDFS when files are grouped 3-of-6 (half the group width) and 12-of-15 (double the group width).

For our experiments, we benchmark throughput for different read sizes in the manner described in Sec. 4.6, and set stripe width for each read request size to optimize for cluster throughput and IO efficiency. For example, we use 1-wide and 6-wide stripes for 8-MB and 48 MB requests respectively.

Okapi provides high throughput at low byte overhead. Fig. 5a shows total cluster throughput with and without decoupling for each access pattern. For the configurations we examine, Okapi observes up to 80% increase in throughput and never performs worse than coupled 6-of-9 for any request size; without sacrificing any space efficiency or durability. Okapi performs the same as 6-of-9 coupled for only 48 MB requests, and same as 3-of-6 coupled for only 24 MB requests which is expected since Okapi's decision to stripe 6-wide and 3-wide coincides with the coupled cases' grouping width.

HDFS's poor throughput for smaller read requests sizes is a consequence of coupling data stripes and groups. For



(a) Read-sizes. **(b)** Cluster metrics.

Figure 6: Read performance for a realistic read-only workload. 6a shows the distribution of read-sizes >1 MB and ≤48 MB for erasure-coded files in Google's production storage cluster. 6b shows throughput and seek count for a trace of read requests pulled from the distribution in 6a.

example, a file is generally chosen to be written in 6-of-9 to provide a certain level of space efficiency. However, 6-wide striping for a 4 MB client read suffers from request fan-out and poor IO efficiency as it splits into four 1 MB disk reads, an unfortunate consequence of a decision made with only *grouping* in mind. A naive solution would be to ignore grouping and blindly choose a narrower group width, say 3-of-6. Not only does this increase storage overhead by 2×, but also the 3-wide striping still loses between 6–34% throughput compared to Okapi. Note that coupled 3-of-6 exhibits greater cluster throughput than coupled 6-of-9 for requests up to 24 MB, but eventually performs worse for larger requests since 3-of-6 requires an extra round-trip to Datanodes to retrieve data past the first data stripe, making 6-of-9 more favorable in these cases. If storage capacity dictates files must be grouped wider, say 12-of-15, then the coupled cluster loses up to half its bandwidth. Moreover, note that by decoupling, it is still possible to achieve 12-of-15 space efficiency *while meeting the same IO performance*.

Okapi reduces seeks dramatically. Fig. 5b shows the benefits in cluster's total seeks per second. Okapi promotes user's ability to choose a stripe width that maximizes sequential IO, thereby reducing the spindle usage by as much as 70% between Okapi and coupled 6-of-9. By tuning the stripe width to be suited for smaller sized reads, clients maximize sequential IO and enable disks to spend less time on seeks and more time on data transfer. The drastic reduction in total seeks promotes greater disk bandwidth, even when disk utilization is constant. Moreover, Okapi minimizes the variance in seeks/sec across different read sizes, reducing peak usage by up to 60% which significantly reduces resource provisioning costs.

Okapi lowers client response time. Fig. 5c shows Okapi provides a substantial reduction in median latency compared to both 3-of-6 and 6-of-9. This is a consequence of both improved cluster throughput and minimizing the fan-out effect. A similar, albeit less significant trend is observed in the 95th percentile latencies (Fig. 5d).

Okapi requires no compromises! Okapi does not sacrifice space efficiency or durability for these improvements. For example, to achieve the same read performance while maintaining durability for 16 MB reads in HDFS, the file would

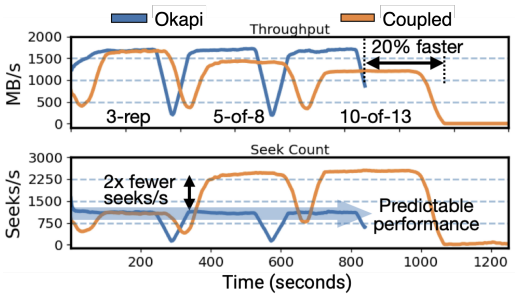


Figure 7: Performance and resource efficiency of file accesses as they transition to wider EC schemes over their lifetimes. Figure shows seeks/s and throughput observed for 16 MB file reads, first when files are in a 3-way replicated format, second in 5-of-8 and then in 10-of-13 EC scheme.

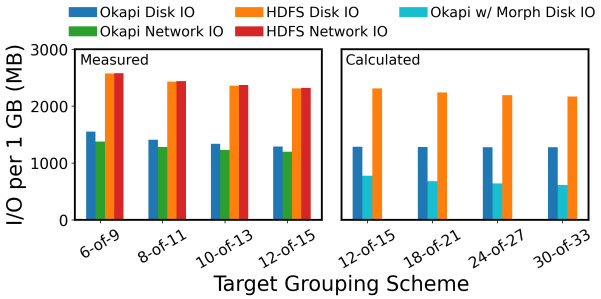


Figure 8: IO cost to transition 1 GB 6-of-9 files to varying target EC schemes. Left shows *measured* disk and network IO cost to transition to wider grouping schemes and then back to 6-of-9. Right shows *calculated* disk IO cost to transition to wider schemes.

need to be written in 2-of-5, which has worse space efficiency than 2-way replication. On the other hand, to achieve the same read performance while maintaining space efficiency in HDFS, the file would need to be written in 2-of-3, which has insufficient durability. Even when comparing against coupled 3-of-6, which has the same space efficiency as 2-way replication, most access patterns still perform noticeably worse than when decoupled. A coupled architecture simply cannot achieve the best of both worlds as Okapi can by decoupling these independent decisions.

Okapi enhances performance for a realistic workload. In this experiment, we measure the read performance in a testbed environment for a workload that uses the read-size distribution of EC files in a production storage cluster at Google as shown in Fig. 6a. We compare cluster performance in decoupled and coupled architectures while executing the same set of random-reads. In this scenario, we use 64 client threads each performing 10K reads. All files conservatively use a 6-of-9 grouping, although today's average EC group is typically wider for space efficiency. Since IO efficiency decreases for mid-sized requests with wider groups, we expect that in practice, the coupled case would actually perform worse than shown in our experiments for this workload.

Fig. 6b shows throughput, seeks/s, and end-to-end run completion time for a realistic read-only workload. Okapi shows a 55% increase in sustained throughput (MB/s) and 65% fewer number of total seeks (35% decrease in seeks/s) due to smarter striping decisions. Note that most read requests in Fig. 6a are smaller than 12 MB, a range of read-sizes that finds the most benefit from decoupling as seen in Fig. 5a. Despite both clusters running identical traces, Okapi's clients completed their work 36% faster than the HDFS's clients. The increase in throughput and decrease in end-to-end run-time shows how a decoupled storage cluster could handle both bursty and extended client traffic better than a coupled architecture.

Okapi maintains IO efficiency despite EC transitions. Morph [29] improves EC transition overheads but overlooks the impact on resource efficiency of accesses to such files. Accesses to same files post-transition to wider EC schemes result in poor IO efficiency. Unlike coupled systems, Okapi remains unaffected by EC transitions. We duplicate their experiment of file EC lifetime transitions from replication to 5-of-8 to 10-of-13 and measure throughput and seeks consumed for 16 MB read accesses to 25000 files in each phase of the file's lifetime. Okapi is able to achieve better throughput, uses 2× fewer seeks/s and achieves predictable performance and resource efficiency as shown in Fig. 7. This issue is further exacerbated in Disk Adaptive Redundant storage clusters, since even hot files can be transitioned to wider schemes, which directly impacts the latencies of user facing applications.

Okapi's benefits improve with median seek distance. The experimental setup has each disk filled with about 315 GB out of 1 TB of disk space. As the disk becomes more full, the median rotational and positioning latency would likely increase, further emphasizing the impact of each seek and making the reduction of seeks-per-second even more significant to cluster throughput.

6.3 Decoupling reduces EC transition IO

Redundancy transitions are common and have varying degrees of urgency and bandwidth/compute requirements. This subsection quantifies this benefit for three major sources of transitions and then experimentally compares Okapi's re-grouping to a coupled system's read-re-encode-write (RRW).

1. Making groups wider as data cools. Large storage clusters perform hundreds-of-thousands of transitions per day. Most of these transitions increase group widths to increase space efficiency for data that has cooled (in terms of access frequency). Okapi can reduce the IO required for such transitions by up to 50%. For example, for 100K 30 MB file transitions from 6-of-9 to 30-of-33 in a day, RRW would require 6.3 PB of total IO ($5 \times 6 \times 100K = 3$ PB of data read, 3 PB of data write and 300 TB of parity write). Regrouping in Okapi only requires 3.3 PB IO for the same transitions.

2. Urgent transitions to address emergencies. Sec. 3 describes an emergency bulk transition situation at Google caused by a sudden increase in the failure rate of a particular disk model. To avoid future reconstruction IO overload, all high-k data on those disks were transitioned to use narrower groups ($k \approx 50$ to $k \approx 15$).

Suppose 66% of the data was in ≈50-wide groups on

≈90K disks in the cluster.[4] Disks had an average capacity of 8 TB and capacity utilization of ≈75%. Transitioning to a 15-of-18 scheme using RRW requires 784.08 PB of transition IO (356.4 PB of data read[5], 356.4 PB of data *re-write*, and 71.28 PB of parity write). Assuming a 100 MB/s sustained disk bandwidth, with a 5% IO budget for EC transitions, 90K disks can transition at the rate of 90000×5 = 450000 MB/s = 0.43 TB/s. It would require over 21 days to transition all 784.08 PB without violating the IO budget. Even if we commission the entire cluster to perform transitions, with each disk issuing transition IO at 100 MB/s, it would take over 1 entire day! In contrast, Okapi incurs an IO traffic of 427.68 PB (45% less) for transitioning via regrouping, i.e. 12 days to complete without violating the 5% IO budget, and <14 hours if the entire cluster was commissioned.

3. Adapting group widths to disk failure rates. Disk-adaptive redundancy (DARE) systems use file transitions to maximize space efficiency based on observed disk failure rates, saving ≈ 20% of disk capacity [25–27]. As with other transition causes, Okapi reduces the IO required.

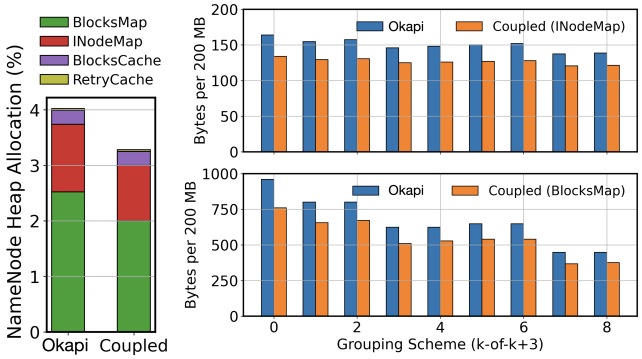
We simulate DARE for a production storage cluster (Backblaze [2]) over a period of 6 years to evaluate the transition costs of re-grouping versus RRW. Data is arranged into files, represented by an ordered array of redundancy groups, and the policies for reacting to failures and AFR changes were adopted from previous work [25]. When a group is marked for transition, the file that it is part of (and hence all groups part of the file) are transitioned together as described in Sec. 4.5.

The simulation showed that Okapi reduces mean disk transition IO cost by approximately 38%. This reduction is lower than earlier examples because it includes IO for block relocation to ensure that all blocks within each group are distributed across different failure domains even after transitions (Sec. 4.5). This overhead can be avoided by being cognizant of such potential re-groupings during file creation, and by placing blocks of adjacent groups of a file across different failure domains. We did not make any assumptions about failure domains in our simulation, placed blocks of a group randomly assuming each disk to be a separate failure domain.

Further, storage clusters typically rate limit background tasks like file EC transitions to prevent interference with foreground IO operations. However, these IO limits are sometimes violated for urgent transitions during risk of data loss. Assuming a 5% daily IO limit for EC transitions [26], we observe that RRW approach results in almost twice as many disks violating this IO cap when compared to Okapi.

Measured transition savings in Okapi. Okapi's regrouping approach for redundancy transitions is more efficient than the traditional RRW approach used in HDFS. In this experiment, we transition a 6-of-9 1GB file to wider groups and then back to 6-of-9. Fig. 8 (left) shows disk and network IO incurred in the two systems as measured in our experimental cluster setup.

[4] Although we cannot give the exact numbers, these are representative.
[5] 90000×8×0.75×0.66 TB = 356.4 PB



(a) Heap percentage **(b)** Byte overhead per file.

Figure 9: Calculated (a) total Java heap allocation overhead in the Namenode for decoupling 200 MB 6-of-9 files and measured (b) byte overhead of 200 MB files in the BlocksMap (bottom) and INodeMap (top) for increasing group widths. Metadata overhead decreases with increasing stripe and group width, eventually converging at very wide groups in both data structures.

Limited by available cluster size, we calculate results for transitions to wider groups analytically. Fig. 8 (right) shows calculated disk IO traffic for transitioning the same 6-of-9 1GB file using Okapi versus RRW.

We observe almost 50% decrease in disk and network IO across all transitions with Okapi. This decrease stems primarily from reduction in bytes written to disk. Note that Morph sees the same transition IO traffic as coupled in Fig. 8. Although Morph uses Convertible Codes to reduce IO overheads by computing new parities from old parites, it only helps when cell-striping is disabled; otherwise, re-striping the file necessitates rewriting it. Okapi overcomes the need to re-write file data, which can be combined with Morph to further reduce transition overheads by up to 70%.

Okapi uses slightly more disk than network IO. Okapi schedules new parity encoding on a target Datanode, so some new parities need not traverse the network. HDFS's RRW implementation instead does this on a client, in part because it must re-write the file data in addition to the new parities.

Okapi shows memory and compute savings. We compare memory and CPU utilization for the Namenode and client by performing 20 concurrent 200 MB EC transitions. With RRW, the client buffers file data in memory before writing to DFS and then clears its cache. With regrouping, the client uses no extra memory to make an RPC call. In Okapi, CPU utilization does not exceed 5.7% compared to 13.5% in HDFS, with ≈75% reduction of average CPU utilization.

6.4 Quantifying decoupling overhead

Namenode metadata memory overhead. We measure the memory footprint of decoupling by analyzing Java heap dumps on the Namenode process in Okapi and HDFS with 6-wide striped, 6-of-9 grouped 200 MB files. Fig. 9a depicts the metadata difference in total heap allocation. HDFS, by default, allocates 2% of the total Java heap to the BlocksMap

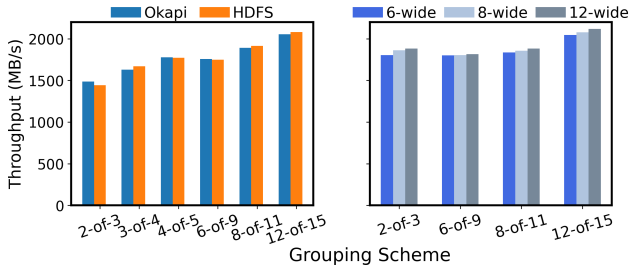


Figure 10: Measured (left) write throughput of files with stripe width equals group width in Okapi and HDFS and (right) decoupled write throughput of files in Okapi with varying stripes and EC schemes.

and 1% to the INodeMap. Okapi compensates for the increase in block objects by allocating 2.52% and 1.22%, respectively, for a collective heap usage increase of 0.74%. This increase allows Okapi to maintain the same number of objects and files in the cluster as conventional DFS with less than a 1% increase in overall Java heap usage.

Fig. 9b presents the memory footprint in the Namenode data structures per file. Okapi uses less than 1 KB per file in the BlocksMap and 170 B per file in the INodeMap, a respective 26% and 22% increase from their former values. These values shrink with wider groups as there are fewer blocks per file and, therefore, less overhead per block object. For example, if files were grouped 12-of-15 as opposed to 6-of-9, the overhead decreases to 20% and 18% respectively. These measurements reflect a reasonable increase in metadata per-byte which only improves with wider grouping schemes.

Minimal impact of decoupling on write throughput. Fig. 10 (left) shows write throughput across various EC schemes when stripe width equals group width. Despite the slight additional complexity in Okapi's decoupled write logic, we do not observe any noticeable decrease to write throughput between HDFS and Okapi.

Fig. 10 (right) compares write throughput with different stripe widths across varying EC schemes. There is no noticeable difference in performance as a direct result of decoupling. 6-of-9 grouping achieves the same write throughput with both 6-wide (coupled) and 8-wide striping (decoupled) despite the fact that 8-wide striping uses partial parities to compute the final parities. This result is consistent with Okapi's implementation, which guarantees the same number of linear operations to compute the final parities, regardless of stripe width.

Further, we observe a small increase in write throughput across all EC schemes as a result of increasing stripe width. A wider stripe can leverage a greater degree of parallelism when writing large amounts of data. However, we find that the difference in throughput across EC schemes can be primarily attributed to the amount of write amplification.

Client write memory overhead. Okapi uses partial parities to avoid buffering copious amounts of data in memory to write a complete group (Sec. 4.3). Fig. 11 shows the memory overhead when striping 6-wide with varying grouping schemes in a decoupled case and when striping/grouping 6-wide in HDFS. With decoupling, naive buffering of client data to eventually compute final parities requires clients to buffer up to 150 MB at 20-of-23. With partial parities, clients buffer at most an acceptable 25 MB.

In fact, the write memory overhead of Okapi with partial parities eventually converges with and passes (i.e., less memory required) coupled schemes beginning from 25-of-28 grouping. In conventional HDFS, the entire stripe of data must be buffered, even for extremely large groups, so the memory overhead scales with group width. With partial parities, however, the overhead is capped at $r \cdot \text{block size} + 1$.

However, when stripe width is greater than group width, in certain instances, it is more cost-effective to opt out of partial parities. This is because buffering all unprotected data cells is more economical than buffering partial parities of size $r \cdot$ block size. Okapi can dynamically determine at runtime whether to employ partial parities based on striping and grouping configuration.

Degraded mode read overhead. We show, both analytically and empirically, that degraded read overheads are minimal. Fig. 12 (left) quantifies the average read amplification and (mid) the average number of disks read during degraded-mode reads using 12-wide groups across varying stripe and read request sizes (assuming that reads can be made at any cell-aligned file offset with equal probability).

Case (1) when stripes are wider than groups. The read amplification is minor (3.23% worse on average for 18-wide stripes; and 4% *better* for 24-wide stripes). Although it may appear that the expected number of disks read is higher, the cause is the wider stripes(a conscious choice by file writer)—the number of *additional* disks read in degraded-mode is very small (2.73% more disks than coupled 18-wide groups).

Case (2) when groups are wider than stripes. Although read-amplification is higher in Okapi, probability of degraded-mode reads is proportionately lower (since fewer disks are accessed). The impact on total cluster throughput is hence commensurately lower. In addition, the required data is fetched from fewer disks (implying fewer seeks and higher IO efficiency), further amortizing the impact on cluster throughput.

Fig. 12 (right) experimentally compares single-failure degraded read latency for 6-of-9 EC files for Okapi (3-wide and 12-wide stripes) with coupled system (6-wide stripes). For small-sized (1 MB) and large sequential (96 MB) degraded-mode reads, Okapi observes no hit in performance. For mid-sized requests (24 MB), Okapi performs 33% worse with 3-wide stripes due to reading 2× more data. Note, however, that degraded reads occur half as often with 3-wide stripes than with 6-wide stripes. Okapi performs 16% worse with 12-wide stripes, because it is a poor stripe width for the given (24 MB) access pattern—it performs worse than 6-wide striping even for 24 MB non-degraded mode reads (see Fig. 5). Worst-case read amplification, and hence tail latency, may be worse in some striping configurations though the average read am-

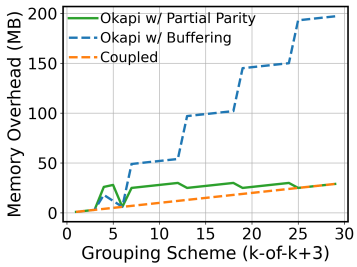


Figure 11: Maximum memory overhead during file creation for varying group widths. For coupled, we set stripe width = group size (k). For decoupled, we set stripe width = 6.

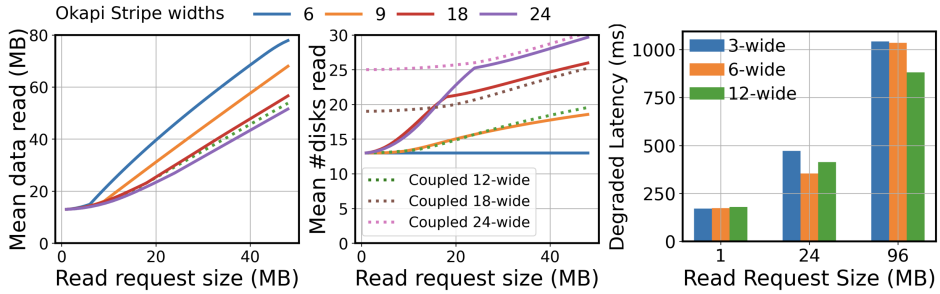


Figure 12: Expected total data read (left) and disks accessed (mid) during degraded reads for group width=12 and varying stripe sizes, with single block failures. The dotted lines refer to coupled configurations. Measured degraded read latency (right) for 6-of-9 files across stripe widths for three read sizes. Okapi's degraded-reads for 24 MB are 33% worse with 3-wide (but occurs half as often), 16.6% worse with 12-wide (due to poor stripe width choice).

plification remains consistent. Benchmarking the stripe/group choices (Sec. 4.6) can help avoid unacceptable overheads.

6.5 Choosing group and stripe widths

This section follows the benchmarking process from Sec. 4.6 and presents performance results for 12 MB and 24 MB read requests under different cluster loads, for degraded-mode, and for different variances in request sizes. Based on our experiments, we recommend a stripe width of $(\lceil \frac{\text{median read request size}}{\text{block size}} \rceil)$ for optimizing cluster throughput and median latency. However, it may be sub-optimal for tail latency in some cases, such as with lots of degraded-mode reads. Application engineers must make a subjective trade-off based on which metric (e.g., throughput, median or tail latency) is most critical for their use case.

Fig. 13a shows throughput and 99th percentile tail latency for 12 MB reads to 6-of-9 erasure-coded files for four stripe widths (1,2,3 and 6) under different cluster loads. As described, the recommended stripe width of $\lceil \frac{12}{8} \rceil = 2$ maximizes throughput and minimizes median, p95, p99 tail latencies by reducing disk seeks and improving disk efficiency. For instance, under heavy load, stripe width of 2 yields ≈40% higher throughput than a coupled configuration (stripe width=6) and achieves 6.8% lower 99% tail latency.

Tail latencies grow with more degraded-mode reads, and degraded-mode reads with decoupled stripe/group widths can cause even greater read amplification and higher latencies. Fig. 13a shows throughput and tail latency for 12 MB reads when one of the nineteen disks has failed. Even in this extreme scenario, where 5% of the disks have failed, the recommended stripe width continues to be the best for throughput, median and p95 latencies, but p99 tail latencies are ≈9% worse than coupled. For latency-sensitive applications, the engineer may choose to stripe 6-wide, accepting ≈16.6% lower throughput to avoid a 9% increase to the tail. Another acceptable trade-off might be to use 1-wide stripes, which exhibit 1.6% higher p99 latency than 6-wide stripes, but only incurs 7% hit in throughput compared to 1-wide stripes.

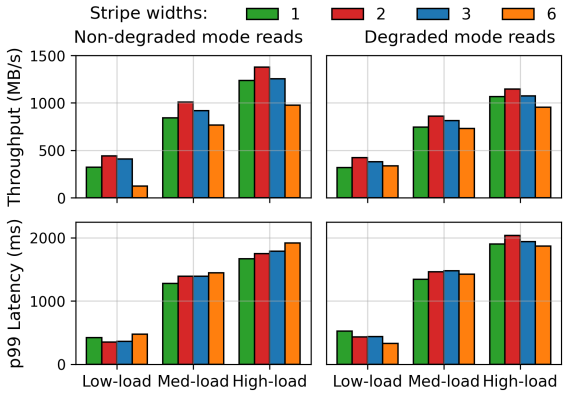
We make similar observations with 24 MB requests in Fig. 13b. As with 12 MB reads, we find that the recommended stripe width of of $\lceil \frac{24}{8} \rceil = 3$ consistently maximizes throughput across load levels. For optimizing p99 latencies, 6-wide stripes generally turn out to be the best.

Fig. 14 shows throughput and latency when request sizes vary under heavy load. We assume request sizes to be normally distributed. We consider two access patterns, 12 MB and 24 MB, and vary standard deviation of request sizes (bounded between 0.5 MB and the file size) to files to study the performance impact. We observe that the recommended stripe widths consistently achieve the highest throughput although the improvements over other widths decrease with increasing variance. While the recommended stripe widths minimize p50 and p95 latencies, the best configuration for p99 latencies can vary depending on the variance in request-size. When request sizes follow a uniform distribution (a rare occurrence, see Sec.3), we observe little difference in performance with different stripe widths.

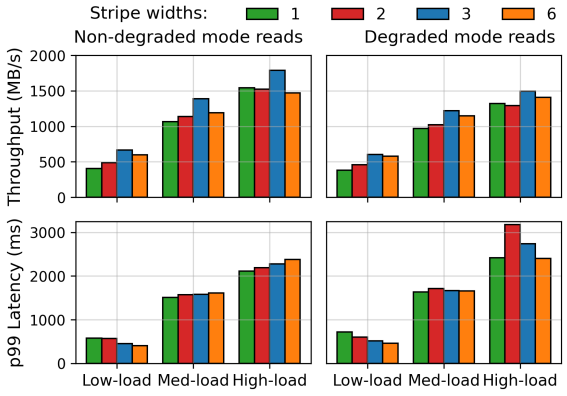
Although we used 8 MB blocks in our experiments, we observe similar trends for block sizes as large as 128 MB. Note that the performance trade-offs will vary depending on cluster setups, software implementations, HDD models, disk capacity utilization, stragglers among many other factors. To find optimal configurations, one must benchmark their applications in their clusters as described in Sec. 4.6.

7 Related Work

Data striping and redundancy grouping have long histories in data storage, beginning in the context of disk arrays [8, 15]. Most disk array configurations couple striping and grouping, such as by using an n-of-$n+1$ RAID5 scheme, with the stripe-unit choice being a target of optimization [9, 44]. One notable exception [19] proposed using parity (grouping) without data



(a) 12 MB reads



(b) 24 MB reads

Figure 13: Throughput (top) and p99 latency (bottom) for (a) 12 MB reads and (b) 24 MB reads to 6-of-9 EC files with stripe widths 1, 2, 3 and 6 for different cluster loads, during normal-mode read operation (left) and during degraded-mode read operation induced by disk failure (right).

striping, an extreme form of decoupling the two.

In both research [1,18,22] and production [13,24,42,47,57,61], DFSs couple stripes and groups. Various research efforts have proposed approaches to tune the striping+grouping configuration based on access patterns and data *-ability goals, given the trade-offs created by the coupling [53,55,58].

Many blob/object stores [24, 38, 41] pack several small objects into a single large file, which is in turn striped/erasure coded across disks. These systems *decouple* the user's view and filesystem's view of how data is stored to avoid small-file inefficiencies. This is orthogonal to Okapi's *decoupling* of stripe and group widths of the resulting large file to improve read efficiency and decrease EC transition overheads.

The prevalent architecture used by hyperscalers (e.g., Google, Microsoft, Facebook) and HPC (e.g., PanFS, Lustre, HDFS) involves the DFS layer managing both striping and grouping. But one could conceivably build a DFS that stripes data across blocks stored in a block-storage layer that separately uses parities to protect groups of blocks. This decouples stripes and groups by having each layer manage one of them. The closest example we can think of is Frangipani-atop-Petal,

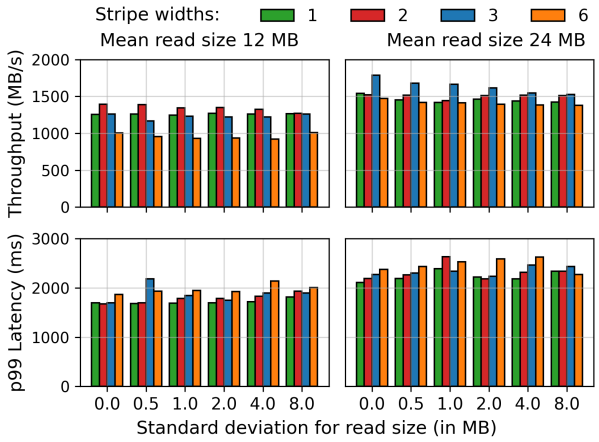


Figure 14: Throughput and p99 latency for reads to 6-of-9 erasure-coded files with different stripe widths as variance in request sizes increases. We consider two access patterns, where requests are normally distributed around (1) 12 MB (left) and (2) 24 MB (right), and vary standard deviation on x-axis.

where Frangipani [52] was a DFS that stored file contents in blocks stored in the Petal [31] replication-based (k=1) block store. However, such an architecture is not adopted by any of the popular DFSs due to its complexity when using EC (k>1) and the fact that it does not allow file-level control for important choices like EC configuration and data placement, all of which are key elements of modern DFS. Okapi retains the benefits of the current DFS architecture while allowing per-file, independent tuning of striping and grouping configurations.

8 Conclusion

Existing DFSs couple data striping and redundancy grouping, resulting in IO performance inefficiencies due to limited options in a complex trade-off space as well as high re-encoding IO overhead . Okapi overcomes these problems by decoupling stripes and groups. The design is simple and adoption-friendly. Experiments confirm significant improvement in IO performance (throughput, seeks and latency) and IO reduction for redundancy transitions.

9 Acknowledgements

We thank our shepherd and the anonymous reviewers for their invaluable feedback. We extend special thanks to Larry Greenfield, Mustafa Uysal, and numerous other researchers and engineers at Google. This research is generously supported in part by NSF grants CNS1956271, CNS1901410, and CAREER 1943409, by a Sloan Foundation Fellowship, and by a VMware Systems Research Award. We also thank the members and companies of the PDL consortium (Amazon, Bloomberg, Datadog, Google, Honda, Intel, Jane Street, LayerZero Research, Meta, Microsoft, Oracle, Oracle Cloud Infrastructure, Pure Storage, Salesforce, Samsung, Western Digital) for their interests, insights, feedback, and support.

References

[1] Thomas E Anderson, Michael D Dahlin, Jeanna M Neefe, David A Patterson, Drew S Roselli, and Randolph Y Wang. Serverless network file systems. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 109–126, 1995.

[2] Backblaze. Disk Reliability Dataset. https://www.backblaze.com/b2/hard-drive-test-data.html, 2013-2018.

[3] Backblaze. Erasure coding used by Backblaze. https://www.backblaze.com/blog/reed-solomon/, 2013-2018.

[4] Eric Brewer. Spinning Disks and Their Cloudy Future. https://www.usenix.org/node/194391, 2018.

[5] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. Technical report, Google, 2016.

[6] Alan D Brunelle. Block I/O layer tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 57, 2006.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[8] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[9] Peter M Chen and David A Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 322–331, 1990.

[10] HG Data. Apache hadoop hdfs insights. *URL https://discovery.hgdata.com/product/apache-hadoop-hdfs*, 2024.

[11] Erasure Code Ceph Documentation. https://docs.ceph.com/docs/master/rados/operations/erasure-code/, 2019.

[12] Facebook. VAST Erasure Coding. https://vastdata.com/blog/introducing-rack-scale-resilience, 2020.

[13] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[14] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2017.

[15] Gregory R Ganger, Bruce L Worthington, Robert Y Hou, and Yale N Patt. Disk arrays: high-performance, high-reliability storage subsystems. *Computer*, 27(3):30–36, 1994.

[16] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[17] Garth A Gibson. *Redundant disk arrays: Reliable, parallel secondary storage*. The MIT Press, 1992.

[18] Garth A Gibson, David F Nagle, Khalil Amiri, Jeff Butler, Fay W Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *ACM SIGOPS operating systems review*, 32(5):92–103, 1998.

[19] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput. In *VLDB*, volume 90, pages 148–161, 1990.

[20] Kevin M Greenan, James S Plank, and Jay J Wylie. Mean time to meaningless:{MTTDL}, markov models, and storage system reliability. In *2nd Workshop on Hot Topics in Storage and File Systems (HotStorage 10)*, 2010.

[21] Rong Gu, Qianhao Dong, Haoyuan Li, Joseph Gonzalez, Zhao Zhang, Shuai Wang, Yihua Huang, Scott Shenker, Ion Stoica, and Patrick PC Lee. DFS-PERF: A scalable and unified benchmarking framework for distributed file systems. *EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-133*, 2016.

[22] John H Hartman and John K Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[23] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for {Wide-Stripe} erasure coding in distributed storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 233–248, 2021.

[24] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[25] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, KV Rashmi, and Gregory R Ganger. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, 2022.

[26] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory Ganger. Pacemaker: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[27] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *USENIX File and Storage Technologies (FAST)*, 2019.

[28] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 1–16, 2023.

[29] Timothy Kim, Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Dax Delvira, Arif Merchant, Gregory R Ganger, and KV Rashmi. Morph: Efficient file-lifetime redundancy management for cluster file systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[30] Spencer Kimball. roadmap: Blob storage. https://github.com/cockroachdb/cockroach/issues/243#issuecomment-73365370, 2015.

[31] Edward K Lee and Chandramohan A Thekkath. Petal: Distributed virtual disks. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92, 1996.

[32] Xiaolu Li, Zuoru Yang, Jinhong Li, Runhui Li, Patrick PC Lee, Qun Huang, and Yuchong Hu. Repair pipelining for erasure-coded storage: Algorithms and evaluation. *ACM Transactions on Storage (TOS)*, 17(2):1–29, 2021.

[33] Chris Mason. Seekwatcher. *URL http://oss. oracle. com/˜ mason/seekwatcher*, 2008.

[34] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, et al. *Monitoring with Ganglia: tracking dynamic host and application metrics at scale*. " O'Reilly Media, Inc.", 2012.

[35] Francisco Maturana and KV Rashmi. Convertible codes: new class of codes for efficient conversion of coded data in distributed storage. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[36] Francisco Maturana and KV Rashmi. Convertible codes: Enabling efficient conversion of coded data in distributed storage. *IEEE Transactions on Information Theory*, 68(7):4392–4407, 2022.

[37] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (ppr). In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.

[38] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm blob storage system. In *Proc. of USENIX OSDI*, pages 383–398, 2014.

[39] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[40] Lluis Pamies-Juarez, Anwitaman Datta, and Frederique Oggier. Rapidraid: Pipelined erasure codes for fast data archival in distributed storage systems. In *2013 Proceedings IEEE INFOCOM*, pages 1294–1302. IEEE, 2013.

[41] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[42] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[43] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in

erasure-coded data centers. *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[44] AL Narasimha Reddy and Prithviraj Banerjee. A study of parallel disk organizations. *ACM SIGARCH Computer Architecture News*, 17(5):40–47, 1989.

[45] Howard Gobioff Sanjay Ghemawat and Shun-Tak Leung. The google file system. pages 1–2, 2003.

[46] Frank Schmuck and Roger Haskin. {GPFS}: A {Shared-Disk} file system for large computing clusters. In *Conference on file and storage technologies (FAST 02)*, 2002.

[47] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[48] Seagate. The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018.

[49] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick PC Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. A survey of the past, present, and future of erasure coding for storage systems. *ACM Transactions on Storage*, 21(1):1–39, 2025.

[50] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The Hadoop distributed file system. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[51] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[52] Chandramohan A Thekkath, Timothy Mann, and Edward K Lee. Frangipani: A scalable distributed file system. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 224–237, 1997.

[53] Eno Thereska, Michael Abd-El-Malek, Jay J Wylie, Dushyanth Narayanan, and Gregory R. Ganger. Informed data distribution selection in a self-predicting storage system. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[54] Zizhong Wang, Haixia Wang, Airan Shao, and Dongsheng Wang. An adaptive erasure-coded storage scheme with an efficient code-switching algorithm. In *Proceedings of the 49th International Conference on Parallel Processing*, pages 1–11, 2020.

[55] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[56] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM / IEEE High Performance Computing Networking, Storage and Analysis (SC)*, 2006.

[57] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[58] Jay J Wylie, Mehmet Bakkaloglu, Vijay Pandurangan, Michael W Bigrigg, Semih Oguz, Ken Tew, Cory Williams, Gregory R Ganger, and Pradeep K Khosla. Selecting the right data distribution scheme for a survivable storage system. Technical report, Technical Report CMU-CS-01-120, Sch. of Computer Science, Carnegie Mellon . . . , 2001.

[59] Qiaori Yao, Yuchong Hu, Liangfeng Cheng, Patrick PC Lee, Dan Feng, Weichun Wang, and Wei Chen. Stripemerge: Efficient wide-stripe generation for large-scale erasure-coded storage. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 483–493. IEEE, 2021.

[60] Mi Zhang, Qiuping Wang, Zhirong Shen, and Patrick PC Lee. Pocache: Toward robust and configurable straggler tolerance with parity-only caching. *Journal of Parallel and Distributed Computing*, 167:157–172, 2022.

[61] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010*, 52, 2010.

Supplemental material for Okapi

The appendix is not peer-reviewed.

A Okapi can be easily adopted by other DFS

Most publicly-described distributed file systems, both open-source and proprietary, that we are aware of couple data stripes and redundancy groups. Okapi's design for decoupling is applicable to and beneficial for all distributed file systems that stripe their files. We describe below the coupled design of a few other contemporary distributed file systems, and show that Okapi's ideas can be adopted into their system.

Colossus File System (Google). The architecture of Colossus File System (Google's distributed file system) is similar to that of HDFS. A $k.r$ erasure-coded file stripe maintains r parity chunks for k (group width) data chunks. The data within the stripe is broken into small (typically 1MB) blocks which are spread across the k (stripe width) disks in round robin fashion [30]. The metadata of the file comprises of the list of file segments, and a mapping of the file segments to chunk locations, which indicate where each of $k+r$ chunks are stored in the storage cluster [45], just like in HDFS. Okapi's design can therefore be directly adopted into Colossus.

Panasas File System. Data is stored in *objects* that are spread across Object Storage Devices (OSDs). Files are striped across objects in a RAID-5 layout. Note that all RAID layouts inherently couple stripes and groups. In RAID-5, one parity chunk is maintained for three data chunks, and each row of blocks is spread across three chunks. Similar to HDFS, per-file metadata stores the storage map. The storage map maintains the striping strategy and location information for each object of the stripe. Okapi's design can therefore be directly adopted into Panasas.

Ceph. The reliability and layout of a file is dictated by EC *profiles* in Ceph. A profile, among other things, is defined by number of data chunks (k), number of parity chunks (m) and the stripe unit size. The stripe units refers to the logically contiguous pieces of data that get spread across the k data chunks in a round-robin fashion.

Unlike HDFS, which *stores* the mapping of chunk to node locations for each group, Ceph uses CRUSH [56], a near-uniform hashing function, to *compute* the locations of the chunks of an erasure-coded group. For an erasure-coded stripe/group of a file, CRUSH uses the file inode number and the index of the stripe (within a file) to compute $k+m$ unique node locations to store the chunks. Each chunk in a group (for durability), as well each chunk in a stripe (for optimal parallelism) must exist on a unique node. Since stripes and groups are no longer the same in Okapi, the algorithm must be modified to ensure node locations are unique across both stripes as well as groups. A naive way to achieve this is to use the existing CRUSH function to find $l = LCM(stripewidth, k+m)$ unique nodes. All chunks in each of the $l/stripe_width$ stripes and $l/group_width$ nodes will then be on unique nodes.



Compass: Encrypted Semantic Search with High Accuracy

Jinhao Zhu
UC Berkeley

Liana Patel
Stanford University

Matei Zaharia
UC Berkeley

Raluca Ada Popa
UC Berkeley

Abstract

We present Compass, a semantic search system for encrypted data that achieves high accuracy, matching state-of-the-art plaintext search quality, while ensuring the privacy of data, queries, and results, even if the server is compromised. Compass contributes a novel way to traverse a state-of-the-art graph-based semantic search index and a white-box co-design with Oblivious RAM, a cryptographic primitive that hides access patterns, to enable efficient search over encrypted embeddings. With our techniques, Directional Neighbor Filtering, Speculative Neighbor Prefetch, and Graph-Traversal Tailored ORAM, Compass achieves user-perceived latencies within or around a second and is orders of magnitude faster than baselines under various network conditions.

1 Introduction

An increasing number of user data systems, including WhatsApp [30], iCloud Backups [10], Telegram [121], Signal [110], and PreVeil [97], have adopted end-to-end encryption because of its strong security properties. In this setting, there has been a rich, decades-long line of work on encrypted search [24, 27, 31–33, 40, 44, 61, 62, 83, 88, 104, 109, 114, 116, 126], which enables the server to search over the encrypted data without learning the data or the query. Despite much progress, two challenges remain: *reduced security* and *low search accuracy*.

Security: The goal is to protect data, queries, results, and access patterns, against a *fully* compromised server. ***i) Leaky practical search.*** A long line of work allows the server to learn some information about the query or data [18, 19, 24, 50, 62, 67, 88, 116], such as search access patterns, to enable fast searches. However, leakage-abuse attacks [17, 55, 64, 74, 94, 96, 129] can reconstruct a significant amount of data or the query from this leakage. ***ii) Search with partial server trust.*** A line of work assumes trusted hardware enclaves at the server [5, 83, 118, 130], but hardware enclaves suffer from a wide range of side-channel attacks [16, 65, 128], including some that can entirely subvert remote attestation and their security [124]. Another line of work [27, 28, 109] assumes that the logical server consists of two or more trust domains, where at least one of them is trusted. However, it has been shown to be very difficult to deploy such different trust domains [26, 73], and attackers can still compromise both servers.

Meaning of social security numbers?

Keyword Search Result: Social Security Number Randomization. On June 25, 2011, the Social Security Administration changed the way Social Security Numbers are issued. Originally Social Security Numbers revealed information about their owners…

Semantic Search Result: Definitions for social security number. Social Security number is the number of a particular individual's Social Security account. Social security number is a nine-digit number issued to citizens, permanent residents, and temporary (working) residents under section 205(c)(2)…

Figure 1: Example of top-1 search results with the keyword search using a popular tool TF-IDF versus semantic search on the MS MARCO dataset. The keyword search result is skewed by the frequency of the phrase 'social security numbers,' and misses the question's focus on meaning, while the semantic search provides a more accurate answer.

Accuracy: Most of the works mentioned above have lower accuracy compared to state-of-the-art plaintext search systems. This is because they only implement the *lexical* search, which is less accurate than the *semantic* search used in modern systems. For example, some works above [3, 24, 27, 61, 62, 83, 114, 123, 126] implement an inverted index that maps each keyword to a list of documents containing that keyword. Such equality matching does not capture the meaning and intent of user queries (e.g., questions). State-of-the-art search systems such as Google products [47], Mac Spotlight [9], Dropbox Dash [120], Bing [81], and Elasticsearch [35] implement semantic search. Semantic search converts both the query and the content into embeddings in a high-dimensional space, which allows the search algorithm to compare the meanings of texts based on their proximity in this space, rather than just matching keywords. As shown in Fig. 1, semantic search is more accurate because it understands the intent and contextual meaning of the query. It also extends search capabilities to unstructured data such as audio, images, and videos.

In this work, through our system ***Compass***, we show that one can achieve the best of both worlds — namely, an efficient and accurate semantic search system without the server-side leakage or trust assumptions mentioned above. Compass

achieves search quality on par with *state-of-the-art plaintext search systems*. Moreover, search in Compass does not leak access patterns and does not rely on trusted hardware, while still protecting against a malicious server-side attacker (§4.8). Compass is designed for scenarios where a user wishes to privately search over their own confidential data, stored in encrypted form at a logical server. In this setting, Compass also enables secure retrieval over an encrypted RAG (Retrieval-Augmented Generation) [69] database.

The few prior proposals on secure embedding search are either inefficient or have weak security guarantees. For example, HERS [36] performs a linear scan using FHE over the entire data, while SANNS [20] combines heavy cryptographic tools like FHE, distributed ORAM, and Garbled Circuits. Other proposals suffer from weaker security assumptions, such as leaky search [11, 132], or relying on non-colluding servers [109]. Tiptoe [51] targets public databases and is thus unsuitable for private data scenarios. Specifically, its core homomorphic techniques depend on data being public to achieve efficiency. We elaborate further in §7.

To realize our goal of high accuracy, Compass aims to execute a powerful *state-of-the-art* semantic search index over encrypted embeddings. Graph-based algorithms currently have state-of-the-art accuracy on diverse workloads [12, 112], and Hierarchical Navigable Small World (HNSW) [79] is a top-performing and commonly-used index of this type [93, 98]. HNSW structures data as a multi-layer graph, with fewer nodes and edges at higher layers. The nodes at each layer are a subset of the layer below, with the bottom layer containing the entire dataset. Search starts from the top layer, traversing the graph using a greedy walk to find the local nearest neighbor, and then continues to the layer below.

Such complex graph traversal in the embedding space would be inefficient to perform with fully homomorphic encryption (FHE) [42] or GarbledRAM [77]. Instead, we build Compass on Oblivious RAM [90], specifically Ring ORAM [101]. Beyond access pattern protection, ORAM offers additional attractive properties for our system. With ORAM, Compass separates the computation from the storage: the embeddings and index are stored on the server in encrypted form, while the search algorithm runs locally on the client. The client interacts with the server only to retrieve the data needed during the search. Fig. 2 shows Compass's architecture. Unlike FHE, this allows us to support any distance metric, enabling a wealth of state-of-the-art embeddings. On the practical side, the efficiency of ORAM has improved significantly since it was first proposed. Additionally, supporting integrity checks on Ring ORAM is straightforward.

While the idea of combining a graph-based index with ORAM is alluring, the main challenge lies in efficiency. HNSW, like other graph-based ANN search algorithms, is optimized for efficient multi-hop traversal of a locally stored graph. A strawman solution that makes a remote ORAM request for every node visited during a search is highly inefficient due to *high bandwidth usage* and *excessive round trips*. Evaluating each candidate node requires fetching embeddings for all of its neighbors and computing their distances to the query. In HNSW, each node typically has tens to hundreds of neighbors. To achieve good accuracy, HNSW usually evaluates tens to hundreds of candidate nodes. As a result, data for thousands of nodes must be fetched remotely, and even with batching ORAM requests, it still requires tens to hundreds of round trips for a single query.

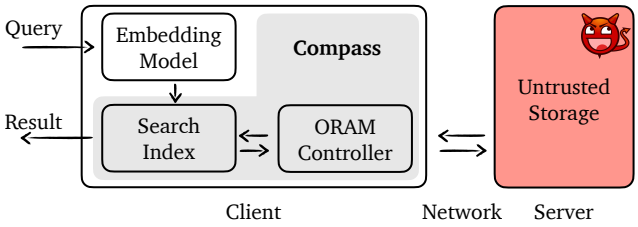


Figure 2: Compass Architecture

To address this challenge, we develop a new encrypted search index that *co-designs* a graph walk through embedding space and an ORAM backend for efficient graph search. Specifically, we introduce three techniques. We introduce *Directional Neighbor Filtering* to reduce bandwidth consumption. The key idea is, at every node in the graph walk, to only fetch the embeddings of a subset of neighbors that are in the same "direction" in the embedding space as the query. Compass determines this subset based on a small chunk of data, which is orders of magnitude smaller than the original dataset, cached locally on the client. Second, we introduce *Speculative Neighbor Prefetch*, which speculatively prefetches additional nodes that are likely to be visited in the future to reduce the total number of round trips required. Finally, we introduce *Graph-Traversal Tailored ORAM* that integrates the search index walk into a white-box "rearrangement" of the ORAM protocol, further improving the communication overhead and significantly reducing user-perceived latency.

Evaluation summary: We implemented and evaluated Compass on 4 popular datasets of different sizes and dimensions. Our results show that Compass's encrypted search matches the accuracy of the state-of-the-art plaintext semantic search algorithm across all datasets. Compass delivers up to 920× speedup over HSNW running on top of ORAM without our techniques and is orders of magnitude faster than two other evaluated baselines. Compass achieves user-perceived latencies of 0.57-1.28 seconds on a slow cross-region network across these datasets. In terms of memory consumption, client-side memory remains small (e.g., 5.5 MB, see Tab. 4) for personal data search scenarios. However, Compass is not yet scalable for global-scale web searches, as client-side memory usage increases to 0.5 GB (Tab. 4), a limitation we leave for future work. Overall, Compass makes encrypted search with high semantic accuracy practical for the first time without compromising security.

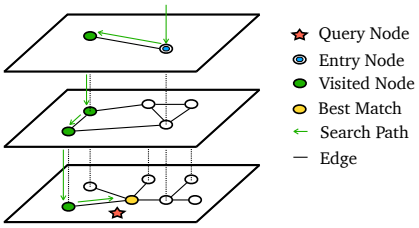


Figure 3: HNSW Graph

2 Background

2.1 Graph-based ANN Search

Similarity search is a core component of large-scale retrieval tasks. Commonly, entities, such as images, user profiles, audio, text, or products, are embedded into high-dimensional spaces with a chosen distance metric (e.g., Euclidean distance). The similarity search task then reduces to the nearest neighbor (NN) search problem: given a dataset D of points in a metric space, a query point q and a target result size K, we must efficiently retrieve the K closest points to q from D.

Since exact NN search is intractable on large, high-dimensional datasets [21, 106] and many applications can tolerate some accuracy loss, practical systems use *Approximate-Nearest-Neighbor (ANN) search*. These methods build indices over the dataset to support high search accuracy and efficiency. Proposed ANN algorithms include tree-based methods [13–15, 25, 53, 76, 85, 111], hashing-based methods [6–8, 43, 46, 54, 56, 71, 75, 92, 119], quantization-based methods [41, 48, 58, 59, 68], and graph-based methods [4, 38, 57, 79, 80, 86]. While each class of algorithms presents trade-offs between accuracy, efficiency, indexing time, and space footprint, graph-based algorithms have emerged as a popular choice due to their state-of-the-art accuracy on diverse workloads [12, 112]. ANN algorithms that do not use data-dependent traversal in the embedding space, like LSH [43, 54, 109], typically achieve lower accuracy with high-dimensional embeddings [12, 112].

HNSW [79] is a widely-used, top-performing graph-based index [12, 112]. It builds a hierarchical, multi-layer graph that has bounded node degrees. The HNSW search algorithm follows a simple iterative greedy search procedure. As shown in Fig. 3, the search begins from a pre-defined entry point. This entry point is chosen during the index construction and is a node on the uppermost layer. The search iterates over each layer, performing a greedy search, before dropping down to the layer below to continue its search. On each upper layer, the greedy search uses a dynamic candidate list of size one and greedily chooses a single node, which then becomes the entry point to the next layer below. Finally, upon reaching the bottom layer, the algorithm performs a greedy search using a dynamic candidate list of size ef and selects K nodes instead of just one. The parameter ef is typically larger than K and controls the trade-off between quality and efficiency, with higher values of ef corresponding to higher quality at the cost of higher search latency. In practice, one can choose ef by empirically generating the accuracy-efficiency trade-off curve over a benchmark dataset and choosing the ef value that meets the application's target accuracy at minimal search latency.

2.2 Ring ORAM

Oblivious RAM (ORAM) [45, 90, 116] allows a client to access data on remote storage without leaking access patterns. Path ORAM [116] is a practical, tree-based ORAM scheme that organizes server-side storage as a binary tree of roughly $\log N$ levels, where each node (bucket) contains Z blocks. Each block contains either real data or a dummy. A path is defined as the sequence of buckets from the root of the tree to one of the leaf nodes. Each block is randomly mapped to a path and resides either on that path or in the stash, a client-side buffer. The client also maintains a mapping of each block to its assigned path, which is called the position map.

We chose Ring ORAM [101], a follow-up to Path ORAM, for Compass because Ring ORAM has low bandwidth consumption (i.e., constant online bandwidth through the XOR trick). Even in a large client storage setting, as noted in [101], Ring ORAM remains competitive with other hierarchical ORAM schemes [29, 117]. Each bucket in Ring ORAM contains $Z+S$ blocks, where S is the number of slots reserved for dummies. Blocks in a bucket are randomly permuted before being written to the server, and the permutation is stored as metadata on the server. Ring ORAM's operations are:

- ReadPath(l,a): Unlike Path ORAM, Ring ORAM reads only one block, the target block a or an unused dummy, from each bucket on the path l. A block becomes invalid once it is read, which is reflected in the metadata.
- EvictPath: Eviction occurs every A ReadPath operations. The evicted path is selected in a reverse lexicographic order.
- EarlyReshuffle: A bucket must be reshuffled after S accesses because it contains only S slots for dummies. This operation reads the bucket from the server, shuffles it, and writes it back to the server.

To access a block a in Ring ORAM, the client first checks the position map to find the path l. Next, it reads the metadata of all buckets on l and uses this information to invoke the ReadPath operation. Thus, Ring ORAM requires two round trips per access. The online bandwidth is $O(\log N)$ and can be reduced to $O(1)$ via XOR operations by the server.

3 System Overview

Our system setup consists of a logical server and multiple clients. Each client owns a unique set of documents and an individual index, both stored on the server. Compass focuses on the case of a personal user searching their confidential data in the cloud, such as on Dropbox [120] or iCloud [9]. Data

sharing could be implemented by adding shared data to each relevant user's index. We do not focus on data sharing and leave optimizations to future work.

Compass's APIs are as follows:

- INIT($\mathcal{D}$): Given a set of documents $\mathcal{D}$, initialize the index.
- SEARCH(Q, K) $\rightarrow$ LIST(ID, SCORE): Given Q, the query embedding, return the top K relevant documents as a list of document IDs and their relevance scores.
- INSERT($\mathcal{F}$): Given a document $\mathcal{F}$, generate the embedding set $\mathcal{E}$ and store both the encrypted document and its embeddings on the server. The size of $\mathcal{E}$ depends on the document length and the model's context window.
- DELETE($\mathcal{F}$): Remove the document $\mathcal{F}$ and its embeddings.

Search, insert, and delete in HSNW are (poly)logarithmic in the number of documents, assuming the HNSW graph approximates a Delaunay graph [79]. Compass preserves the complexity of these operations. File updates are handled by deleting the old file and inserting the updated version. Large files are split into equal-sized chunks based on the embedding model's context length. Each chunk is treated as a separate file with its own embedding. For efficient ORAM initialization, one can use the bulk loading protocol from BULKOR [72].

3.1 Threat Model & Security Guarantees

Our system is based on a two-party model: a client and a (logical) server. The logical server may consist of multiple physical machines. The server is untrusted and can be maliciously compromised. Unlike some prior works that weaken the threat model by assuming multiple logical servers with at least one being honest [27, 103, 109, 125], we assume the entire server is untrusted. We also do not rely on uncompromised hardware modules or hardware trust assumptions at the server, like TEE-based systems [39, 83]. This means that the server in Compass can deviate from the protocol arbitrarily. Compass ensures that the server cannot learn any information about the user query, such as the query embedding, the IDs and lengths of the returned documents, or whether the query matches a previous one ("query access patterns"). Moreover, the server cannot learn the data contents from the search index or modify the search results without detection by the client.

Compass does not hide the type of operation, whether a user is performing a search, insert, or delete, on remote data. We do not protect against timing side-channel attacks based on the client's execution duration or the server's observation of timing patterns in user requests. We consider the system configuration parameters listed in §3.1.1 as public information. The server can roughly tell how large the data of each user is (e.g., how many GBs each user has), but Compass does not reveal to the server the size of query results. (Of course, these can be protected through padding [63] in time and space at a performance cost.) This strong model means that Compass protects against a plethora of attacks [17, 55, 64, 74, 94, 96, 129] on encrypted search that leverage access patterns or volume of results. Like much prior work [51, 83, 126], Compass does not protect against DoS attacks from the server. The server may drop the requests from clients at any time or even delete the client's data. We assume it is the server's business interest to maintain a good availability to attract more customers.

The client, who owns the outsourced data, is trusted with its own documents. That is, the client may search and read the contents of these documents. However, the client cannot access or perform a search on the documents of other clients, even if the client colludes with the server.

1. The challenger $\mathcal{C}$ chooses a uniformly random bit b.
2. The adversary $\mathcal{A}$ chooses two equally large datasets D_0 and D_1, as well as one set of public parameters param. The challenger initializes the system with D_b and param.
3. The adversary iterates as follows. At step i:
   (a) The adaptive adversary $\mathcal{A}$ chooses a pair of requests $q_{i,0}$ and $q_{i,1}$, which can be search, insert, or delete.
   (b) When the challenger receives a pair of requests, it first checks whether the type matches. If they are of different types, the challenger aborts. Otherwise, the challenger interacts with the adversary in the role of the server according to the Compass's protocol for request $q_{i,b}$. As part of this execution, the client protocol verifies the integrity of the data from the server. If any verification fails, the challenger aborts.
   (c) The challenger gets the request result r_i.
4. The adversary $\mathcal{A}$ outputs a guess b'.

Figure 4: Security game for Compass

3.1.1 Security Definition

Like prior work [107, 115], we provide an indistinguishability-based security definition. In our security game, there is a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ who acts as the client and the server of Compass, respectively. According to the threat model, the challenger is honest and the adversary is malicious, which means $\mathcal{A}$ can deviate from the protocol arbitrarily. The attacker wins the security game if it can either learn partial information about the query or data, or modify the search results without detection by the challenger (client).

Let param denote the set of public parameters in our system, including HNSW parameters M and ef, the number of cached layers, the size of the directional filter *efn* (§4.5), the size of the speculating set *efspec* (§4.6) used in our modified HNSW search. We define the security game in Fig. 4. The adversary $\mathcal{A}$ wins the game if the challenger $\mathcal{C}$ does not abort and one of the following two conditions is met: ***1)*** $b' = b$; ***2)*** the sequence of queries $(q_{i,b}, r_i)$ is an incorrect execution of plaintext Compass's search algorithm with param on D_b.

Theorem 1. *Assuming a collision-resistant hash function and an IND-CCA2 secure encryption scheme, for any probabilistic polynomial time stateful adversary $\mathcal{A}$, $\mathcal{A}$'s chance of winning in the security game in Fig. 4 instantiated with Compass's protocol is: less than half plus negligible for winning condition (1) and negligible for winning condition (2).*

Due to space constraints, we show a proof sketch in §4.10, leaving the full proof in the supplementary material §C.

4 Compass's Search Index

We now describe Compass's index. While we build Compass on top of HNSW, our techniques are generally applicable to other graph-based ANN indices. We primarily focus on the search since the insert algorithm in HNSW closely resembles that of the search, and the techniques described in this section apply to both. A summary of notation is shown in Tab. 1.

4.1 Data Layout

Before we dive into the details of each technique, we present Compass's data layout. The client holds:

- *ORAM-related data*: a stash for overflow blocks, a position map that maps each block to a path, metadata for each bucket, and a cached treetop for the ORAM tree (§4.7).
- *HNSW-related data*: metadata for HNSW search, including the dimension of the embeddings, the number of layers, and the degree bound M. If client-side caching (§5) is enabled, the client also stores the graph for the upper layers and the embeddings of nodes within these layers.
- *Quantized hints* that are a part of our Directional Neighbor Filtering technique, which we explain later in §4.5.

The server holds the ORAM tree in the Ring ORAM construction. Each node (bucket) of the tree contains $Z+S$ blocks. In our search index, each ORAM block corresponds to one node in the HNSW graph. The block corresponding to node x stores the embedding of x and its neighbor list, which consists of identifiers for the neighbors of x. Each bucket has a tree of hashes for the integrity check (§4.8).

4.2 Workflow overview

Fig. 2 shows the architecture and workflow of our system. Similar to a plaintext search on embeddings, the user's query is first transformed into a dense, fixed-dimensional vector using a pretrained embedding model. This query vector is then used to search within the embedded document corpus. Our search framework builds on HNSW and provides security guarantees by storing vector embeddings and the graph index encrypted on a remote server, accessed by the ORAM controller. During the search process, the client interacts with the server to retrieve the index and embeddings.

Symbol	Description
M	degree bound of traversed nodes in HNSW search
ef	size of dynamic candidate list in HNSW search
efn	size of directional filter
$efspec$	size of speculation set
Z	maximum number of slots for real blocks per bucket
S	number of slots reserved for dummies per bucket
A	eviction rate

Table 1: Summary of Notation

4.3 An initial attempt

We now present a natural way of traversing HNSW using ORAM: the client follows HNSW's search algorithm, making an ORAM request for each node visited. This solution is inefficient as it requires substantial bandwidth and numerous round trips. This inefficiency arises because HNSW is designed for local, multi-hop searches, while ORAM turns each node access into a non-trivial client-server request. It nevertheless establishes the foundation and terminology for our subsequent improvements in the rest of this section.

This HNSW greedy traversal is illustrated in Fig. 5a, which we now explain. Starting from the entry node (1), the algorithm fetches all its neighbors and adds them to the *candidate list*, namely, it visits them. We say that the algorithm *visits* a node if it retrieves the ORAM block of this node. In the subsequent iteration, the node in the candidate list closest to the query is selected as the next candidate node. The unvisited neighbors of this candidate are then visited and added to the candidate list. In Fig. 5a, it takes 7 search iterations to find the nearest node, with every node in the graph being visited. This algorithm works well when computation and storage are colocated. However, in our setting, the client has to retrieve neighbors' embeddings and neighbor lists from the remote server through ORAM requests, making it inefficient because of multiple costly network round trips and significant data transfer. In HNSW, the number of required round trips depends on ef, and each node's degree is bounded by M. Therefore, a search requires $ef \cdot M$ rounds of communication to retrieve the embeddings and neighbor lists of $ef \cdot M$ nodes. Empirically, in the final layer of HNSW, both M and ef range from tens to hundreds for sufficient accuracy, causing high search latency and bandwidth usage.

4.4 Towards Compass's search index

To understand our approach, it helps to first consider an attempt to reduce the number of sequential ORAM requests in the strawman above. The idea is to include the embeddings and neighbor lists of the neighbors in the neighbor list of a node within the ORAM block of that node. This will halve the number of ORAM requests and round trips as compared to the strawman. However, this strategy causes $M\times$ storage overhead on the server side because each node's embedding would

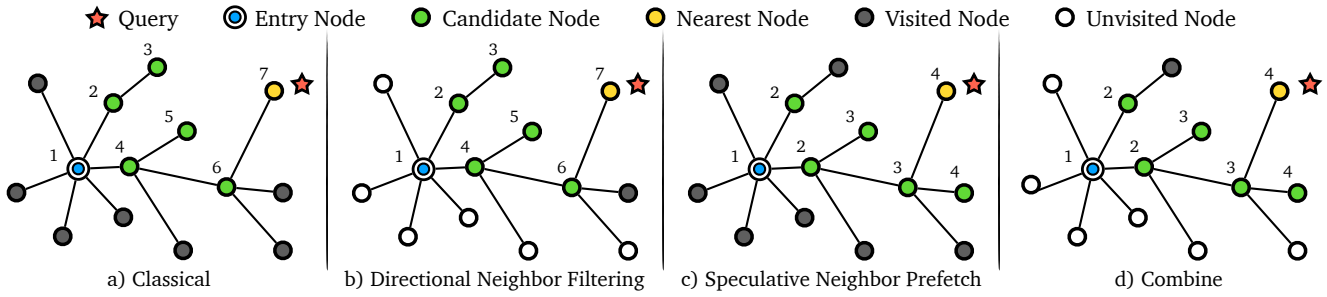


Figure 5: HNSW graph traversal followed by our ORAM-friendly traversal techniques. The search aims to identify the Nearest Node to the Query. The distance here is Euclidean distance. The numbers indicate iterations in the traversal, and which nodes are *processed* at each iteration. We say that a node is processed, if its neighbors are visited.

be stored both in its own ORAM block and in the blocks of its neighbors. Moreover, the increase in storage does not result in substantial bandwidth savings due to the larger ORAM block size.

Instead, we propose a novel way to traverse the HNSW graph designed for running on top of ORAM. Our traversal approach significantly reduces round trips and bandwidth usage via three techniques. The first two techniques make black-box usage of ORAM; *Directional Neighbor Filtering* (§4.5) reduces the network bandwidth overhead and *Speculative Neighbor Prefetch* (§4.6) reduces the number of network round trips required. Our third technique, *Graph-Traversal Tailored ORAM* (§4.7), makes white-box use of Ring ORAM to leverage the memory access characteristic in graph search to further reduce bandwidth and computation overhead.

4.5 Directional Neighbor Filtering

The key idea behind our Directional Neighbor Filtering technique is to consider only a subset of a node's neighbors during traversal because those closer to the query point are more likely to contribute to the final result; namely, to embed a sense of "direction".

One possible way to implement this idea is to store with every node, not only its own embedding but also the list of *compressed* embeddings of its neighbors. The intuition is to determine which compressed neighbors are closer to the query, and only fetch those from the ORAM, reducing bandwidth consumption. One can use compression techniques like PCA [60]. However, since a node does not have a large number of neighbors, we found that the compression is not effective in reducing the size of each ORAM block because there are not enough neighbors to amortize the space taken by the compression state (e.g., the transformation matrix).

Instead, we compress together *all the nodes* into a data structure we call *Quantized Hints*. The client stores the Quantized Hints, mapping each node ID to its quantized embedding. We find Product Quantization (PQ) [58] fits well here: it is a widely-used quantization method that is highly effective at compressing a large number of high-dimensional vectors. For example, we achieve 98% compression on various datasets (see §6.3.3) with a codebook size equivalent to 128 full coordinates, thus requiring a minimal amount of client storage. However, if we use PQ directly (namely, construct the index directly on quantized embeddings), there is a big accuracy drop because the closest quantized neighbor to the query is often not the closest full-coordinates neighbor.

Instead, our idea is to use the quantized nodes merely as "directional hints": for each node, we identify the top *efn* closest neighbors to the query based on the hint, and then fetch their *full* coordinates (the ground-truth) from the server to identify the next node to process. The intuition is that the closest *efn* quantized neighbors are very likely to contain the closest (full-coordinates) neighbor. By fetching only *efn* out of M neighbors, this technique reduces bandwidth usage by a factor of $\frac{M}{efn}$. We demonstrate in §6.4 that, with an appropriately chosen *efn*, the impact on search accuracy is negligible.

For clarity, we include an example of using directional neighbor filtering during the search in Fig. 5b. Assume we fetch only the top 2 closest neighbors in each iteration. At the beginning of a query, the Compass client computes its quantized embedding. Then, during each iteration in the traversal, let node A be the currently closest node to the query in the candidate list. The client iterates through A's neighbor ids, looking up their quantized embedding in Quantized Hints, and computing each neighbor's distance from the query in the quantized space. The client then only fetches the top $efn = 2$ closest neighbors from the server in a batched ORAM request (§4.7), and adds them to the candidate list. Compared to Fig. 5a, we can see in Fig. 5b that some of the neighbors of the nodes processed at (1), (4), and (6), which are not among the top 2 closest nodes to the query, are not visited. We thus avoid visiting 6 nodes in the graph, which approximately halves the bandwidth consumption.

4.6 Speculative Neighbor Prefetch

In speculative execution [66], the processor makes guesses of potentially useful instructions and executes them in advance to improve performance. Our Speculative Neighbor

Prefetch shares the same intuition. Each time the Compass client performs a round trip to the ORAM server to retrieve nodes that will be visited next by HNSW's search, the client also speculatively fetches additional nodes. This implicitly requires batching the access of multiple ORAM blocks into one request, which is introduced in §4.7. These additional nodes are not yet needed by the HNSW search algorithm, but are likely to be visited later. By retrieving these *likely-needed* nodes alongside the *currently-needed* node, we can reduce the number of future network round trips to ORAM during search. But how do we identify these nodes? Fortunately, the candidate list used in the greedy search algorithm is a good basis for speculation. The candidate list is sorted by the distance to the query. In our algorithm, we fetch the first *efspec* nodes' neighbors within one request. Once the response is received from the remote server, the candidate list is updated by evaluating the neighborhood of each node in the speculative set. With this technique, Compass reduces the number of round trips by a factor of *efspec*.

In Fig. 5c, we show an example of Speculative Neighbor Prefetch. In each iteration, the client extracts two candidates from the candidate list and visits their neighbors simultaneously. In this toy example, Compass reduces the number of iterations from 7 to 4. We show in §6 that implementing this technique on real datasets significantly reduces the search steps required while maintaining the same accuracy.

DiskANN [57] uses a similar speculative approach in the different setting of optimizing the interaction between memory and SSD. In DiskANN, they fetch a small number of extra nodes to balance the cost of computing and SSD bandwidth. However, in Compass, the cost model of fetching data from the disk and fetching data from ORAM over the network is different. In our setting, a key observation is that optimizing the number of round trips is much more important than optimizing bandwidth because the round trips directly affect user-perceived search latency and the bandwidth consumption is already in a reasonable ballpark. Therefore, depending on the dataset, for Compass, the size of the speculation set *efspec* can grow as large as 16 to achieve good overall performance.

Fig. 5d shows the final example of the search process when we compose both Directional Neighbor Filtering and Speculative Neighbor Prefetch. Compared to Fig. 5a, the algorithm in Fig. 5d achieves the same search results with three fewer network round trips and six fewer node retrievals.

4.7 Graph-Traversal Tailored ORAM

To reduce the cost of an HNSW traversal, or more generally a graph traversal, Compass optimizes how a graph is stored in ORAM and rearranges how ORAM requests are performed.

As mentioned in §4.1, each block in our ORAM stores both a node's embedding and its neighbor list. We now motivate this choice. Recall that graph storage consists of two parts: coordinates and neighbor lists. In each step of a greedy search, we first fetch the current candidate's neighbor list and then fetch the coordinates based on the neighbor list. A simple approach is to use two ORAMs, one for coordinates and another for neighbor lists. However, this requires two network round trips per search step, assuming that we batch the request for coordinates. Our design is based on the observation that the neighbor list belongs to a node whose coordinates we have already obtained in previous steps. By storing a node's coordinates and neighbor list in the same block, we fetch the neighbor list early to avoid an extra network trip. Let D be the dimension of the embedding and M be the degree bound. The bandwidth increases from $M*(D+1)$ to $M*(D+M)$. Typically, the degree bound is much smaller than the embedding dimension, so the increase is roughly 1.2 to 2×.

We build our system based on Ring ORAM and we make several white-box modifications to make it more efficient under graph traversal workload. The reader should recall the Ring ORAM background in §2.2. Each bucket's metadata is cached on the client to avoid extra round trips. As mentioned earlier, we use batching, a technique already shown to be effective [107, 126, 133], to reduce the number of round trips for parallel requests in the same search step. In Ring ORAM, evictions are scheduled after every A path access to keep the stash within a reasonable size. With batching, evictions are delayed if the batch size is larger than A.

Compass introduces ***multi-hop lazy eviction***: as the Compass client walks from hop to hop through the ANN graph to answer a query, the client performs a sequence of ORAM request batches without performing their associated eviction, responds to the user, and finally performs eviction. The advantages of multi-hop lazy eviction are substantial. In ORAM schemes, online bandwidth refers to the cost of reading data from the server, which affects response time, while offline bandwidth covers other costs like evictions. By delaying eviction until the end of a query, users experience lower search latency. This method works especially well in Ring ORAM, where offline costs are higher than online costs. For example, as mentioned in [101], for a set of properly selected parameters, the offline bandwidth is roughly $2-2.5\times$ smaller than the online bandwidth, and the online bandwidth is further improved by a factor of $\log N$ with the XOR trick. The downside of lazy eviction is that the buckets near the root run out of dummy budget quickly, leading to more frequent early reshuffles. We mitigate this with classic tree-top caching [101]: by caching the top t levels of the ORAM tree, we can delay 2^t evictions and keep the reshuffle rate reasonable.

With batching and multi-hop lazy eviction, the number of requests per batch and the number of batches per query introduce additional leakage. In a graph, the number of neighbors for each node can vary, which affects batch sizes, and the speed at which the greedy search converges can also be different, which affects the number of batches. We address this issue through a co-design with the graph algorithm. The main idea is to keep the number of search steps the same at each

layer of the HNSW graph and pad each node to have the same number of neighbors. We explain the details in §4.9.

Stash analysis: Another concern with delayed eviction is its impact on stash size, which can indirectly affect security. During a query, the client keeps all fetched blocks in the stash until the final eviction. As part of the eviction process, blocks from multiple delayed paths are fetched from the server. With a larger stash, eviction becomes inefficient if we perform a linear scan to find the candidate block for a specific bucket. To address this, we sort the blocks in the stash by their assigned path ID. This allows us to recompute the range of path indices for a target bucket and quickly locate candidate blocks in the stash. Insertion complexity increases to $O(\log N)$, where N is the size of the stash. However, it reduces the search cost for a bucket of size Z from $O(ZN)$ to $O(\log N)$, as the potential blocks are already grouped by path ID during sorting. We prove a stash bound for Compass's adaptation of Ring ORAM. In §B, we show that after the eviction is complete after the query, the size of the stash is upper bounded by the size of the stash in the original Ring ORAM.

4.8 Malicious server protection

A malicious server can alter ciphertexts, rearrange the layout of ORAM buckets, perform replay attacks, or provide incorrect query responses. To protect integrity and freshness, we follow the idea from the Path ORAM literature [102, 115] and construct a Merkle tree on top of the ORAM tree. If we label the buckets linearly, the hash of a non-leaf bucket is defined as:

$$\mathsf{H}_i = \mathcal{H}(\mathsf{B}_i || \mathsf{H}_{2i} || \mathsf{H}_{2i+1}), \tag{1}$$

where B_i is the "snapshot" of the i-th bucket, and H_{2i} and H_{2i+1} are hashes of the child buckets. For leaf buckets, the hashes of child buckets are omitted. In Path ORAM, B_i is simply the concatenation of all blocks in the bucket:

$$\mathsf{B}_i = b_1 || b_2 || ... || b_Z, \tag{2}$$

In Ring ORAM, however, each access retrieves only one block per bucket along the path. To avoid fetching the remaining blocks, we construct a secondary Merkle tree within each bucket, where the j-th leaf node is the hash of the j-th block in the bucket. Now, B_i is the root hash of the secondary Merkle tree of the i-th bucket. During the ReadPath operation, for each bucket, we retrieve $\lceil \log(Z+S) \rceil$ hashes from the secondary tree to reconstruct B_i, plus an additional hash for its sibling bucket. For EarlyReshuffle and EvictPath, since we already read Z blocks from each bucket, we only fetch S hashes of the remaining blocks. Note that for EarlyReshuffle, since we reshuffle individual buckets instead of entire paths, we must also retrieve the hashes of the target bucket's child buckets and the hashes of sibling buckets of its ancestors.

Algorithm 1: SEARCH($q, ep, ef, efspec, efn$)

Input: query q, entry point ep,
size of speculation set $efspec$,
size of directional filter efn,
number of nearest to q elements to return ef
Output: ef closest neighbors to q

```
1  V ← ep // set of visited nodes
2  C ← ep // set of candidates
3  W ← ep // set of found nearest neighbors
4  n ← ⌈ef/efspec⌉
5  for step ← 0...n do
6      E1 ← extract top efspec nearest nodes from C to q
7      E2 ← ∅
8      foreach e1 ∈ E1 do
9          foreach e2 ∈ neighbors(e1) do
10             if e2 ∉ V then
11                 E2 ← E2 ∪ e2
12     t ← efspec * efn
       // Based on quantized hints
13     E3 ← extract top t nearest nodes from E2
14     oram.batch_access(E3, t)
15     foreach e ∈ E3 do
16         V ← V ∪ e
17         C ← C ∪ e
18         W ← W ∪ e
19         if |W| > ef then
20             remove furthest element from W to q
21 return W
```

4.9 Putting it All Together

With the techniques above, we present our search algorithm for each layer in Alg. 1. In HNSW, the insertion is the same as the search, but it uses a different ef. To delete a node, the standard way is to search for the node and mark it as deleted.

Compared to vanilla HNSW, the main difference lies between line 4 and line 14. The number of search iterations, n, is determined by $\lceil ef/efspec \rceil$, as we extract the top $efspec$ nodes from the candidate list as the speculative set, E_1, in each iteration. In the first iteration, there is only one candidate, the entry point, so only one node is extracted from the candidate set. We then traverse the unvisited neighbors of the nodes in E_1 (referred to as E_2), ranking them based on their distance to the query, estimated using quantized local hints. The top t nodes from E_2 are selected as E_3, and a batch ORAM request is issued to retrieve their full coordinates and neighbor lists. The value of t is set to $efspec \cdot efn$, as we want efn neighbors per candidate. If the total number of nodes in E_3 is less than t, the batch ORAM request is padded to t for security. Once the full coordinates are retrieved, we insert each node in E_3 into the set of visited nodes V, the set of found nearest neigh-

bors W, and the candidate set C. The set W is dynamic, as the furthest element will be removed when its size exceeds ef. After completing n iterations, W is returned, and the first K nodes in W are the final search result of q. For search in upper layers, the ef is set to 1, and therefore we only apply directional neighbor filtering.

4.10 Security proof sketch

We provide a proof sketch covering the salient points, leaving the full proof in the supplementary material §C.

The integrity guarantee of our protocol comes directly from the Merkle tree on top of ORAM, which has already been proposed and proved [37, 102, 115]. Therefore, the attacker must respond correctly to all requests, with their only capability being to learn which items are accessed on the server during these requests. Prior works have already shown that batching of ORAM requests doesn't reduce security [126, 133].

Compass differs from these works in that our ORAM requests follow a multi-hop pattern. Each query in Compass triggers multiple batches of ORAM accesses, followed by lazy eviction at the end. For each type of query, the number of batches and the number of paths accessed per batch are determined by the public search parameters (see Alg. 1). During lazy eviction, the number of eviction paths depends only on the total number of earlier accesses, so it is also query-independent. Because the paths are chosen randomly, Compass does not leak the access pattern under the threat model described in §3.1. Different query types (e.g., search and insert) use different ef, so an adversary can still distinguish the operation type. As noted in §3.1, Compass does not try to hide which type of operation is being performed.

4.11 Fault Tolerance Considerations

In case of failure, Compass handles recovery similarly to prior work on ORAM fault tolerance [23]. At a high level, the client (1) persists its local state to disk and (2) periodically uploads this encrypted state to the server as checkpoints. At each checkpoint, the server stores a complete copy of the ORAM tree. Between checkpoints, the client prepares each ORAM operation (including access, reshuffle, and eviction) by persisting the associated information to the local disk. Specifically, it records the type of operation and the offsets of buckets and blocks that will be accessed on the server. Similarly, the server logs each ORAM operation before execution. If the client crashes, it first restores its local states from the latest checkpoint and then re-executes all ORAM operations performed after that checkpoint. This step is necessary to prevent the leakage of access patterns. Otherwise, if the same query is executed before and after the crash, the server could infer information because the accessed paths would be identical. Note that Compass fully protects against a malicious server only if the client-side disk remains intact, allowing recovery of local states and operation logs from disk. If the client's disk also fails, recovery relies solely on the server-provided state and logs. In this scenario, a malicious server might provide stale client states or an outdated ORAM tree, neither of which the client could detect.

5 Implementation

We implemented Compass in approximately 5k lines of C++ code. [1] We use the Faiss [34] library for HNSW construction and Product Quantization. We use AES-256-CBC to encrypt ORAM blocks and SHA-256 for hashing, via OpenSSL's EVP [2]. To save the cost of frequently visited nodes in upper layers, we cache all but the last two layers of the HNSW graph locally on the client. We set the number of search steps in the second-to-last layer to 1. Due to space constraints, we explain the reason behind the choice of cached layers and search steps in §A. We show the client-side overhead associated with layer-wise caching in §6.3.3. We follow the instructions from Ring ORAM [101] and choose $Z = 32$, $S = 64$, and $A = 36$ for our ORAM configuration. We use a slightly larger value for S and a smaller value for A compared to the bandwidth-optimal parameters recommended in [101] to reduce early reshuffles and the number of round trips.

6 Evaluation

In this section, we aim to answer the following questions:

- What is the search quality of Compass, and how does it compare to state-of-the-art plaintext search systems?
- What is the overhead introduced by Compass, and how does it compare to prior encrypted search schemes?
- How does Compass compare to an approach where the client downloads and stores the entire index locally?

6.1 Experimental Setup

We perform our experiments on the Google Cloud Platform with one n2-standard-8 instance (8 vCPUs and 32 GB memory) as the client and one n2-highmem-64 instance (64 vCPUs and 512 GB memory) as the server. We use Linux Traffic Control (tc) [1] to simulate network conditions. Following prior work [20, 109], we evaluate on two sets of network configurations to simulate same-region (fast) and cross-region (slow) networks. Specifically, the fast network has 3 Gbps bandwidth and 1 ms round-trip latency, and the slow network has 400 Mbps bandwidth and 80 ms latency.

We evaluate our system on four popular datasets widely used for ANN search: MS MARCO [22], TripClick [100], SIFT1M [58], and LAION [108]. The specifications for each dataset are presented in the upper half of Tab. 2. We build the HNSW graph with $M = 64$ for LAION and SIFT1M, and

[1] https://github.com/Clive2312/compass

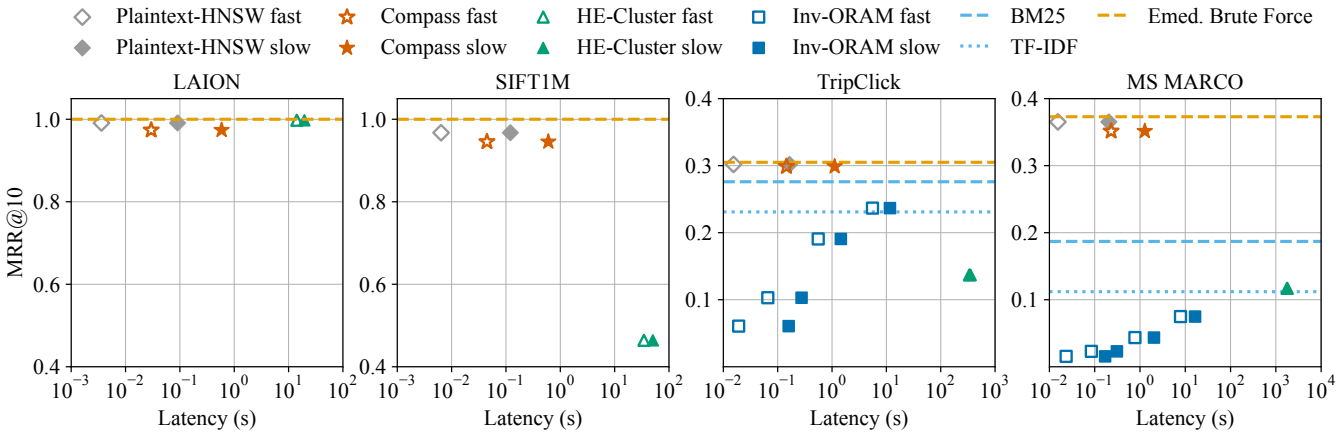


Figure 6: Comparison of Compass's search quality versus latency with Baselines. For TripClick, the results of the HE-Cluster under two networks are overlapped as the main bottleneck is the computation on the server. The search latency for HE-Cluster on MS MARCO is extrapolated due to memory limitation. (See estimated memory consumption in §6.3.3)

	MS MARCO	TripClick	SIFT1M	LAION
dim	768	768	128	512
# docs	8,841,823	1,523,871	1000,000	100,000
# queries	6,980	1,175	10,000	1000
ef	48	36	20	10
efspec	8	6	4	2
efn	24	24	12	12

Table 2: Specifications and parameters for each dataset.

$M = 128$ for TripClick and MS MARCO. For SIFT1M, each embedding is divided into 8 sub-vectors for PQ. For the other higher-dimensional datasets, we split the embeddings into 32 sub-vectors. The search parameters, shown in Tab. 2, are set to achieve a Recall@10 of at least 0.9.

6.2 Baselines

We compare our system with three baselines:

Inv-ORAM is a secure lexical search: an inverted index stored within ORAM [83, 126]. The inverted index maps each keyword to the documents that contain it. Relevance scores are computed using TF-IDF [99], a widely used algorithm in lexical search systems. Each *(keyword, document, score)* pair is stored inside an ORAM block. To avoid leakage of query length and keyword frequency, queries are padded to the longest query length (12 for TripClick and 16 for MS MARCO). Ideally, to hide keyword frequency, we would fetch the maximum number of documents a keyword maps to. However, as pointed out by [87], this can be worse than streaming the entire database. Therefore, following [83], we truncate each keyword's document list to a fixed size. The list is sorted by relevance score before truncation. To simulate OBI [126], we implement this baseline with batch ORAM access and our improved stash eviction algorithm. We evaluate this baseline on text-based datasets: TripClick and MS MARCO.

HE-Cluster is a secure semantic search based on homomorphic encryption and clustering. Tiptoe [51] demonstrates an efficient way to search for a private query on a *public* database by leveraging clustering to avoid linear communication overhead. This baseline is a natural extension of Tiptoe with the database also encrypted under homomorphic encryption. The clustering technique from Tiptoe directly applies here. However, other optimization methods from Tiptoe do not apply, as they were specifically tailored for a public-data scenario. Following Tiptoe, each dataset of size N is clustered into roughly $\sqrt{N}$ clusters using k-means, and boundary conditions are handled similarly. To prevent leakage of cluster size, each cluster is padded to the same size. The server-side computation for this baseline is parallelized across 64 threads.

Plaintext-HNSW is an insecure semantic search baseline in which the client sends a query to the server, the server executes a vanilla HNSW search on the plaintext data, and then returns the results.

6.3 Performance

We measure performance in two dimensions: search quality and latency. For search quality, we use the mean reciprocal rank at 10 (MRR @ 10), where the reciprocal rank is the inverse of the rank of the first relevant item in the results. For Compass, we report the user-perceived latency under the semi-honest setting. The end-to-end latency results, including those under malicious settings, are provided in §6.3.1. For the Inv-ORAM baseline, we set the truncated document list size to 10, 100, 1000, and 10,000, and report the results separately. The tradeoff is that a larger size results in better search quality but worse latency. In Fig. 6, we present the search performance of Compass along with that of the baselines. We also include the plaintext search accuracy using TF-IDF and BM25 [105], two widely used search algorithms, alongside the accuracy of the brute-force embedding search, which represents the highest

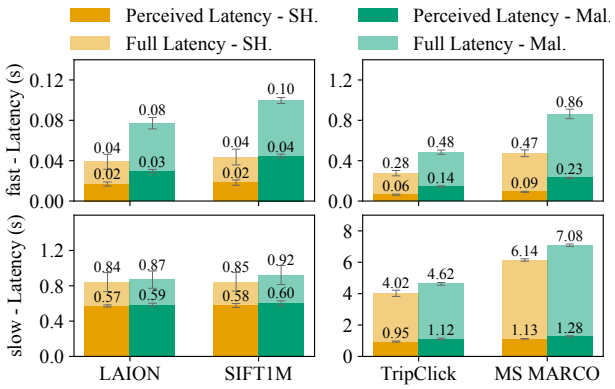


Figure 7: Breakdown of Compass's search latency. *SH* represents semi-honest and *Mal* represents malicious.

achievable accuracy with the current embedding model.

Overall, Compass outperforms both secure baselines in search quality, matching the accuracy of the brute-force embedding search in all four datasets. Compared to Plaintext-HNSW, the insecure baseline, Compass incurs a latency overhead of approximately 6–10×. For Inv-ORAM, the accuracy is affected by the size of the truncated list. Although a smaller truncated list allows Inv-ORAM to achieve lower latency than Compass, its search quality is significantly worse. To match the accuracy of plaintext TF-IDF, Inv-ORAM requires a larger truncated list, which significantly increases latency. The search quality of the HE-Cluster baseline varies across datasets. In the LAION dataset, the HE-Cluster achieves an accuracy comparable to Compass. However, in other datasets, its accuracy drops significantly and falls below that of plaintext TF-IDF on the TripClick dataset. This discrepancy is likely due to the health-related content of TripClick, where text sequence matching is particularly effective in identifying health-related terms. Compared to HE-Cluster, Compass is orders of magnitude faster in latency. While clustering reduces communication costs to sublinear, the server's computation cost remains linear. Although latency can be further improved with parallelization using more CPUs and instances, this is expensive and difficult to scale for multiple users.

6.3.1 Latency breakdown

In Fig. 7, we present the latency breakdown of Compass under various network and security assumptions. We report both user-perceived and full latency. The main contributor to user-perceived latency is ORAM operations. The overhead of distance computation and graph traversal is minimal, requiring less than 0.02 seconds even on MS MARCO. The full latency includes additional time spent on ORAM eviction.

With our lazy eviction technique and the XOR trick from Ring ORAM, the user-perceived latency is reduced by 1.5–5.6× compared to the full latency. For smaller datasets like LAION and SIFT1M on the slow network, the savings are less significant because the bottleneck shifts from bandwidth to round trips, as the graph traversal involves multiple rounds, whereas the eviction only takes two. The cost of integrity verification for malicious security is relatively low compared to the search, especially on larger datasets and slower networks.

Compass focuses on search over private data. For personal use cases, dataset sizes are typically comparable to smaller ones such as LAION and SIFT1M. The full latency for these datasets is under 1 second, even on slow networks. For a large-scale dataset like MS MARCO, the perceived latency is only 1.3 seconds. While eviction takes 5.8 seconds, this process can run in the background, overlapping with other application tasks, such as the subsequent generation step in the RAG pipeline. Overall, Compass achieves practical user-perceived latency across all four datasets and two network settings.

Under a more restrictive network such as 4G (50Mbps bandwidth, 100ms latency), the perceived latency remains acceptable as the primary communication overhead is from eviction. For example, the perceived latency is 0.94 seconds for SIFT1M and 2.03 seconds for MS MARCO in this case.

	Compass				HE-Cluster		Inv-ORAM	
	Per.	Int.	Full	RT	Full	RT	Full	RT
LAION	0.7	0.4	13.5	8	225.5	1	-	-
SIFT1M	1.1	1.0	12.2	8	737.2	1	-	-
TripClick	6.0	3.1	132.8	9	929.2	1	234.5	1
MS MARCO	8.9	5.1	226.4	9	2262.5*	1	300.1	1

Table 3: Comparison of communication cost (MB) and round trips per query. *Per.* represents non-eviction cost and *Int.* represents cost for interity verification. * indicates the result is extrapolated.

6.3.2 Communication

Tab. 3 presents the per-query communication costs and network round trips. For Compass, we additionally report the communication cost before lazy eviction and the cost for integrity verification. For Inv-ORAM, we use a truncated document list size of 10,000 for reasonable search quality. In Compass, the additional communication cost of the integrity check is minimal. Communication costs for LAION and SIFT1M in Compass are similar, even though SIFT1M has 10× more vectors. This occurs because LAION's embedding dimension is 4× larger than SIFT1M's. Compass achieves lower communication costs compared to the HE-Cluster baseline because communication complexity is (poly)logarithmic in Compass but $O(\sqrt{N})$ in HE-Cluster. Additionally, the FHE scheme used by HE-Cluster produces larger ciphertexts. Compared to Inv-ORAM, Compass requires similar communication bandwidth. Note that here we're comparing the total communication cost for a query. As we mentioned in §6.3.1, the main communication overhead in Compass comes from lazy eviction. For the non-eviction communication cost, which affects

	Embed. Size	Graph Size	Compass Server		Compass Client				HE-Cluster		Inv-ORAM	
			SH.	Mal.	Hints	ORAM	Graph	Total	Server	Client	Server	Client
	GB	GB	GB	GB	MB	MB	MB	MB	GB	MB	GB	MB
LAION	0.19	0.12	0.95	0.98	3.55	1.89	0.05	5.49	8.75	0.8	-	-
SIFT1M	0.48	0.51	6.19	6.69	7.75	27.87	0.21	35.84	15.6	0.71	-	-
TripClick	4.37	1.49	24.19	24.69	47.25	29.86	0.37	77.48	245.09	4.48	12	605.64
MS MARCO	25.33	8.55	193.50	197.50	270.58	225.99	2.07	498.65	1345*	11.14*	12	809.16

Table 4: Comparison of memory consumption. * indicates the result is extrapolated.

user-perceived latency, Compass requires less than 10 MB, even for the largest dataset, MS MARCO.

While both baselines are single-round protocols, Compass requires multiple rounds per query. We justify this for two reasons: First, Compass aims to match the quality of plain text search, and the additional round-trip overhead adds less than one second, even in slower network settings. In high-latency networks, the number of round trips can be further reduced by tuning the search parameters at the cost of a minor accuracy loss. Second, our Speculative Neighbor Prefetch limits the increase in round trips as dataset size grows. For example, MS MARCO and TripClick require the same number of round trips, despite MS MARCO having 6× more entries.

Compass's server is network-bound, primarily used for storage and lightweight XOR operations. In our experiments on the LAION dataset, the server achieves 436 queries per second with 26% CPU utilization under a 32 Gbps bandwidth and 25 concurrent client threads continuously issuing requests.

6.3.3 Memory Consumption

We report the breakdown of memory consumption in Tab. 4. The second and third columns show the sizes of plaintext embeddings and the HNSW graph for reference. We report the server-side memory consumption under both malicious and semi-honest settings. On the server side, Compass requires 3.2–6.8× more memory compared to storing the embeddings and the graph in plaintext. The main overhead here comes from ORAM, as a certain amount of dummy blocks is necessary for a smaller stash and fewer reshuffles. On the client side, the memory overhead consists of three parts: quantized hints, ORAM-related states (including the position map, metadata, and cached tree-top), and the cached layers of the HNSW graph. The main contributors to client-side memory usage are the quantized hints and ORAM-related states. Since the graph size in HNSW decreases exponentially as we move up the layers, the memory overhead for caching these layers is minimal. The size of the quantized hints is approximately 1% of the size of the original embeddings. Among the ORAM-related structures, metadata accounts for 60–80% of the overhead. We think this is a reasonable trade-off as we save 2× round trips by caching it. For a large-scale dataset like MS MARCO, Compass requires approximately 500 MB of client memory. For smaller, user-scale datasets, memory consumption is below 100 MB. When client-side memory is limited, Compass can further reduce memory usage by caching fewer layers of the graph and ORAM tree or by using less accurate hints, at the cost of additional round trips and bandwidth to maintain search quality. Compared to Compass, HE-Cluster requires significantly more server-side storage due to the size of HE ciphertexts. For MS MARCO, it needs over 1 TB of memory, which exceeds the capacity of our server instance. As a result, we extrapolated its memory and communication cost. Inv-ORAM requires a lot less server-side memory. This is because instead of storing an embedding vector per document, it only needs to store the IDs of keywords per document. Since passage lengths in MS MARCO and TripClick are usually under 100 words, the storage overhead is smaller.

6.3.4 Insert & Delete

We now briefly discuss how Compass compares to two secure baselines on insert and delete. For HE-Cluster, inserting or deleting a document securely is difficult without streaming the entire dataset; otherwise it could reveal which cluster the document belongs to. In Inv-ORAM, the truncation used for search can also be applied to insertion, and therefore the latency is similar to the search shown in Fig. 6. Deleting a document requires fetching the full list of documents for each keyword, which is expensive. In the HNSW graph, deleting a document is typically done by finding the node and marking it as deleted, which takes a similar time as a search. Insertion works like a search but uses a different candidate list size. For example, with a candidate list size of 160, Compass can insert a document in 19.2 seconds over the slow network and 2.6 seconds over the fast network on the MS MARCO dataset. There is a trade-off between insertion time and search speed. Faster searches usually require more time during insertion to build a better graph by carefully connecting to the best neighbors.

6.4 Ablation Study

We perform an ablation study to evaluate our techniques across all datasets under the slow network. The results are summarized in Fig. 8. Our study consists of three sets of experiments, each focused on evaluating one of our techniques. For Directional Neighbor Filtering, we fix the size of spec-

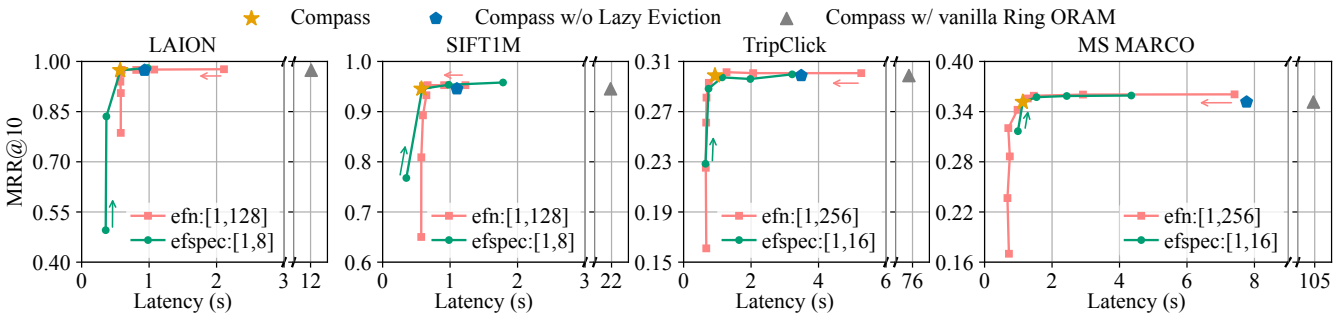


Figure 8: Ablation study on Compass's techniques. Each arrow denotes the direction of decreasing parameter values.

ulation set (*efspec*) to the value in Tab. 2 and vary the size of directional filter (*efn*), incrementally doubling it from 1 up to $2M$. Setting $efn = 2M$ disables this technique. Similarly, for Speculative Neighbor Prefetch, we fix the size of directional filter and vary the size of speculation set from 1 to twice its default value. When $efspec = 1$, Speculative Neighbor Prefetch is disabled. Finally, to evaluate Graph-Traversal Tailored ORAM, we first disable lazy eviction and then further disable batching. For each experiment, we compare the search quality and perceived latency against the default configuration specified in Tab. 2.

In Fig. 8, we observe that initially, a smaller *efn* reduces bandwidth usage by filtering irrelevant neighbors without compromising search quality. However, below a certain threshold, accuracy starts to drop significantly, and latency stops improving. This happens because relevant neighbors start being pruned, and bandwidth is no longer the bottleneck. A similar trend is observed for *efspec*. A very high *efspec* requires guessing more nodes prematurely during the search, leading to a greater impact on accuracy. In practice, the optimal *efn* and *efspec* can be determined by testing a small sample of queries after index construction, jointly with ef.

For Graph-Traversal Tailored ORAM, batching ORAM requests effectively reduces latency by 12–20×, due to far fewer round trips per query. Lazy eviction particularly benefits datasets like TripClick and MS MARCO, which have larger graph degrees and larger speculative sizes. These two datasets require accessing more paths per query, resulting in more frequent evictions if lazy eviction is disabled.

6.5 Comparison with Client-side Index

A strawman alternative to enable search over encrypted data is storing the entire index, including both the graph and embeddings, on the client device. In this setting, the server keeps a copy of the encrypted index, and the client initializes by downloading the entire encrypted index from the server. When the index is updated (e.g., through insertions or modifications), the client uploads these changes in encrypted form to the server. This allows other devices owned by the user, or the same device after disconnection (such as in a web setting), to retrieve the latest index. Uploading the entire new index for every update is expensive, so the client uploads only incremental changes and periodically merges these into a new (re-)encrypted index.

Compared to Compass, this approach allows faster queries after the client's initialization since all data is available locally. However, it requires large storage and memory on the client device, and it incurs a significant cold-start overhead to download the index. Such requirements affect its practicality, especially for users accessing the system via web or mobile devices. We empirically compare Compass with this alternative in terms of client storage and memory, as well as the communication costs associated with the first query.

Storage-wise, this approach requires the client to store the entire index. As shown in Tab. 4, this results in a 27–75× increase in client storage overhead compared to Compass. Moreover, efficient querying requires loading the entire index into memory. Such a large client storage requirement is problematic for mobile devices. According to recent surveys [122], mobile phones typically have around 384 GB of storage with 4 – 16 GB of RAM. Hosting a client-side index significantly restricts available storage and memory, limiting the number of applications users can run simultaneously during queries. Large datasets like MS MARCO, which requires 34GB of memory, cannot even fit directly into the RAM of typical mobile devices. While a separate line of research optimizes graph search between memory and storage [57, 113], it works best with at least several GB of memory.

In the web setting, the overhead of downloading the full index recurs whenever a user starts a new web session. To compare first-query costs, we initialize both systems without any client-side state, requiring each client to first download the state before executing a query. We report bandwidth and latency under the slow network in Tab. 5. For client-side indexing, the main overhead is downloading the index, and the latency increases to 12 minutes for MS MARCO. This initial connection delay is a significant disruption for users every time they have a new web session. In contrast, Compass achieves a consistently low latency of several seconds across all evaluated datasets. Moreover, client-side indexing incurs higher operational costs. For example, downloading the full MS MARCO index has an egress cost of $4.06, whereas Compass incurs only $0.058 for client states. The estimated

	Bandwidth		Latency	
	Client-Index	Compass	Client-Index	Compass
	GB	MB	(s)	(s)
LAION	0.24	6.23	5.2	0.73
SIFT1M	0.97	36.95	22.4	1.45
TripClick	5.84	83.49	121.7	2.84
MS MARCO	33.86	507.52	698.2	12.32

Table 5: Comparison of cold-start query communication cost and latency between Compass and client-side indexing.

cost in Compass (including compute and egress) is $0.01 per query. Each index download in the client-side approach is equivalent to roughly 375 queries in Compass.

Beyond these practical constraints, there are additional conceptual advantages for server-side indexing. First, as personal data volumes grow and embedding dimensions increase, the index will inevitably become too large for storage on any client device. With multimodal embedding models becoming increasingly powerful [127], users will likely include a broader variety of file formats beyond text and images. Furthermore, embedding dimensions are increasing to achieve better semantic understanding. For example, OpenAI's latest text-embedding-3-large model generates embeddings up to 3072 dimensions [89]. Second, according to a survey [27] of end-to-end encryption companies (for file storage, email, and chat), these companies preferred a server-side solution due to concerns about local resource consumption, limited web and mobile compatibility, and the difficulty of indexing large enterprise data volumes locally. In fact, Compass can be applied in the enterprise setting using the proxy model [23, 95, 107], where the company hosts an on-premise Compass client that enforces access control, while a remote server maintains the encrypted index and data.

7 Related Work

Lexical encrypted search: Searching over encrypted data has been a rich and fruitful line of work. Most existing work focused on lexical search, which generally delivers lower search quality compared to Compass. Furthermore, to achieve efficiency, many approaches reduce security in the following ways, whereas Compass does not make these compromises: ***i)*** Leaking access patterns [18, 19, 24, 50, 62, 67, 88, 114, 116], which can be exploited by leakage-abuse attacks [17, 55, 64, 74, 94, 96, 129]. ***ii)*** Assuming trusted hardware [5, 83, 118] that can be exploited by a wide range of side-channel attacks [16, 65, 124, 128]. ***iii)*** Assuming non-colluding servers or at least one trusted server [27, 28, 109].

Encrypted semantic search systems: Encrypted search over embeddings is a nascent line of work. HERS [36] uses FHE to perform a linear scan over embeddings, which results in a high overhead. Designed for the public database setting, Tiptoe [51] performs a more efficient linear scan over embeddings by crucially relying on the data at the server being public, so Tiptoe does not provide a solution for searching over encrypted data (nor for malicious security, integrity protection, and sublinear search) in contrast to Compass. Furthermore, Tiptoe's clustering technique reduces search quality, as discussed in §6. Pacmann [131] is a concurrent work, also designed for the public setting. Pacmann requires $O(\log N\sqrt{N})$ bandwidth and the client has to stream the entire database for preprocessing.

Other works attempt to build sublinear indices over embeddings, but sacrifice security. For example, the index traversal in [11, 132] is not privacy-preserving and leaks query information if the server has some knowledge about the plaintext. Preco [109], Riazi et al. [103], and Wu et al. [125] weaken security by relying on two non-colluding servers. Preco mentions the possibility of removing the non-colluding assumption by adopting a single-server PIR scheme but estimates that the resulting performance would become orders of magnitude slower.

Targeting at a different setting, SANNS [20] and Panther [70] combine several heavy-weight cryptographic tools, such as lattice-based homomorphic encryption, distributed oblivious RAM, and garbled circuits, for database privacy and therefore have higher performance overhead.

The line of work on ***secure inference*** [49, 52, 78, 82, 84, 91] is complementary to Compass; together, they can enable private Retrieval Augmented Generation (RAG) [69] on encrypted RAG data. RAG typically consists of nearest-neighbor retrieval in embedding space followed by model inference. Compass can be used for the retrieval part of RAG, and this line of work for the model inference part of RAG.

8 Conclusion

Compass is a search system over encrypted embedding data that achieves 1) comparable accuracy to the state-of-the-art search algorithm in plaintext, 2) strong security guarantees against malicious attackers, and 3) practical user-perceived latency at low server operation costs. Compass achieves these properties through a novel search index that co-designs the traversal of a graph-based ANN algorithm on top of Oblivious RAM via three techniques.

Acknowledgments

We would like to thank our shepherd, Sebastian Angel, and the anonymous reviewers for their insightful feedback and comments. We also thank Mayank Rathee for his helpful discussions. This work is supported by gifts from Accenture, AMD, Anyscale, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP as well as NSF CAREER 1943347.

References

[1] Linux Traffic Control. https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html.

[2] OpenSSL EVP. https://wiki.openssl.org/index.php/EVP.

[3] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Coeus: A system for oblivious document ranking and retrieval. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[4] Ann Arbor Algorithms. KGraph: A Library for Approximate Nearest Neighbor Search, 2023.

[5] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, 2019.

[6] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 2008.

[7] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, 2015.

[8] Alexandr Andoni and Ilya Razenshteyn. Optimal Data-Dependent Hashing for Approximate Near Neighbors. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, STOC '15, 2015.

[9] Apple. Building a search interface for your app. https://developer.apple.com/documentation/corespotlight/building-a-search-interface-for-your-app.

[10] Apple. iCloud keychain security overview. https://support.apple.com/guide/security/icloud-keychain-security-overview-sec1c89c6f3b, 2021.

[11] Daisuke Aritomo, Chiemi Watanabe, Masaki Matsubara, and Atsuyuki Morishima. A privacy-preserving similarity search scheme over encrypted word embeddings. In *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*, 2019.

[12] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 2020.

[13] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[14] Erik Bernhardsson. annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk.

[15] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, 2006.

[16] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:SGX cache attacks are practical. In *11th USENIX workshop on offensive technologies (WOOT 17)*, 2017.

[17] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015.

[18] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. *Cryptology ePrint Archive*, 2014.

[19] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013*. Springer, 2013.

[20] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M Sadegh Riazi. SANNS: Scaling up secure approximate k-Nearest neighbors search. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[21] Kenneth L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry - SCG '94*, 1994.

[22] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, and Jimmy Lin. Ms marco: Benchmarking ranking models in the large-data regime. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021.

[23] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud.

In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[24] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[25] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, 2008.

[26] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on trusting distributed trust. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022.

[27] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[28] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[29] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[30] Gareth T Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horvárth, and Tibor Jager. Security analysis of the whatsapp end-to-end encrypted backup protocol. *Cryptology ePrint Archive*, 2023.

[31] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. *Cryptology ePrint Archive*, 2019.

[32] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *Advances in Cryptology–CRYPTO 2018*, 2018.

[33] Ioannis Demertzis and Charalampos Papamanthou. Fast searchable encryption with tunable locality. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[34] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[35] Elastic. Semantic search. https://www.elastic.co/guide/en/elasticsearch/reference/current/semantic-search.html.

[36] Joshua J Engelsma, Anil K Jain, and Vishnu Naresh Boddeti. Hers: Homomorphically encrypted representation search. *IEEE Transactions on Biometrics, Behavior, and Identity Science*, 2022.

[37] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Freecursive oram: Nearly free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[38] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 2019.

[39] Benny Fuhry, HA Jayanth Jain, and Florian Kerschbaum. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[40] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*, 2016.

[41] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[42] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, 2009.

[43] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, 1999.

[44] Eu-Jin Goh. Secure indexes. *Cryptology ePrint Archive*, 2003.

[45] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 1996.

[46] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. iDEC: indexable distance estimating codes for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 2020.

[47] Google. What is semantic search? https://cloud.google.com/discover/what-is-semantic-search.

[48] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[49] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. *Advances in neural information processing systems*, 2022.

[50] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. Shadowcrypt: Encrypted web applications for everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[51] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 396–416, 2023.

[52] Xiaoyang Hou, Jian Liu, Jingyu Li, Yuhan Li, Wenjie Lu, Cheng Hong, and Kui Ren. Ciphergpt: Secure two-party gpt inference. *Cryptology ePrint Archive*, 2023.

[53] Michael E. Houle and Michael Nett. Rank-Based Similarity Search: Reducing the Dimensional Dependence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2015.

[54] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, 1998.

[55] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS*, 2012.

[56] Omid Jafari, Parth Nagarkar, and Jonathan Montaño. mmLSH: A Practical and Efficient Technique for Processing Approximate Nearest Neighbor Queries on Multimedia Data. In Shin'ichi Satoh, Lucia Vadicamo, Arthur Zimek, Fabio Carrara, Ilaria Bartolini, Martin Aumüller, Björn Þór Jónsson, and Rasmus Pagh, editors, *Similarity Search and Applications*, Lecture Notes in Computer Science, 2020.

[57] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, 2019.

[58] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 2010.

[59] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus, 2017.

[60] Ian T Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, 2016.

[61] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, 2013.

[62] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[63] Darya Kaviani, Deevashwer Rathee, Bharghv Annem, and Raluca Ada Popa. Myco: Unlocking polylogarithmic accesses in metadata-private messaging. In *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

[64] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[65] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 2020.

[66] Butler W Lampson. Lazy and speculative execution in computer systems. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, 2008.

[67] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. Mimesis aegis: A mimicry privacy Shield–A System's approach

to data privacy on public cloud. In *23rd usenix security symposium (USENIX Security 14)*, 2014.

[68] V. Lempitsky and A. Babenko. The inverted multi-index. IEEE Computer Society, 2012.

[69] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 2020.

[70] Jingyu Li, Zhicong Huang, Min Zhang, Jian Liu, Cheng Hong, Tao Wei, and Wenguang Chen. Panther: Private approximate nearest neighbor search in the single server setting. *Cryptology ePrint Archive*, 2024.

[71] Mingjie Li, Ying Zhang, Yifang Sun, Wei Wang, Ivor W. Tsang, and Xuemin Lin. I/O Efficient Approximate Nearest Neighbour Search based on Learned Functions. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[72] Xiang Li, Yunqian Luo, and Mingyu Gao. Bulkor: Enabling bulk loading for path oram. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[73] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Popa. The deployment dilemma: Merits & challenges of deploying mpc, 2023.

[74] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yuan Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 2014.

[75] Kejing Lu and Mineichi Kudo. R2LSH: A Nearest Neighbor Search Scheme Based on Two-dimensional Projected Spaces. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[76] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. VHP: approximate nearest neighbor search via virtual hypersphere partitioning. *Proceedings of the VLDB Endowment*, 2020.

[77] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *Advances in Cryptology–EUROCRYPT 2013*, 2013.

[78] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive*, 2023.

[79] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[80] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 2014.

[81] Microsoft. Semantic ranking in azure ai search. https://learn.microsoft.com/en-us/azure/search/semantic-search-overview, 2021.

[82] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[83] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[84] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[85] Marius Muja and David G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[86] Gonzalo Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 2002.

[87] Muhammad Naveed. The fallacy of composition of oblivious ram and searchable encryption. *Cryptology ePrint Archive*, 2015.

[88] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014.

[89] OpenAI. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/, 2024.

[90] Rafail Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990.

[91] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[92] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. Neighbor-sensitive hashing. *Proceedings of the VLDB Endowment*, 2015.

[93] Pinecone. Vector search: Hierarchical navigable small worlds. https://www.pinecone.io/learn/series/faiss/hnsw/.

[94] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[95] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, 2011.

[96] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[97] Preveil. Encrypted email and file sharing. https://www.preveil.com/.

[98] Qdrant. An introduction to vector databases. https://qdrant.tech/articles/what-is-a-vector-database/.

[99] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, 2003.

[100] Navid Rekabsaz, Oleg Lesota, Markus Schedl, Jon Brassey, and Carsten Eickhoff. Tripclick: the log files of a large health web search engine. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021.

[101] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[102] Ling Ren, Christopher W Fletcher, Xiangyao Yu, Marten Van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-ram. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[103] M. Sadegh Riazi, Beidi Chen, Anshumali Shrivastava, Dan Wallach, and Farinaz Koushanfar. Sub-linear privacy-preserving near-neighbor search. *Cryptology ePrint Archive*, 2019.

[104] Panagiotis Rizomiliotis and Stefanos Gritzalis. Oram based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, 2015.

[105] Stephen E Robertson and Steve Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR'94: Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, 1994.

[106] Aviad Rubinstein. Hardness of approximate nearest neighbor search. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, 2018.

[107] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[108] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. Laion-400m: Open dataset of clip-filtered 400 million image-text pairs, 2021.

[109] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. Private approximate nearest neighbor search with sublinear communication. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[110] Signal Messenger. https://signal.org/.

[111] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE conference on computer vision and pattern recognition*, 2008.

[112] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*, 2022.

[113] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[114] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy (SP)*, 2000.

[115] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 2018.

[116] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. *Cryptology ePrint Archive*, 2013.

[117] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

[118] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment*, 2021.

[119] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 2013.

[120] Dropbox team. Introducing new tools for the next generation of knowledge work. https://blog.dropbox.com/topics/company/updated-tools-new-plans-and-web-redesign, 2023.

[121] Telegram Messenger. https://telegram.org/.

[122] Efosa Udinmwen. Here's why your smartphone storage is disappearing so fast. https://www.techradar.com/pro/Heres-why-your-smartphone-storage-is-disappearing-so-fast, 2024.

[123] Peter Van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. Computationally efficient searchable symmetric encryption. In *Secure Data Management: 7th VLDB Workshop*, 2010.

[124] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Sgaxe: How sgx fails in practice, 2020.

[125] W. Wu, U. Parampalli, J. Liu, and M. Xian. Privacy preserving k-nearest neighbor classification over encrypted database in outsourced cloud environments. In *World Wide Web*, 2019.

[126] Zhiqiang Wu and Rui Li. OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption. In *30th Annual Network and Distributed System Security Symposium, NDSS*, 2023.

[127] Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, Daliang Xu, Qipeng Wang, Bingyang Wu, Yihao Zhao, Chen Yang, Shihe Wang, et al. A survey of resource-efficient llm and multimodal foundation models. *arXiv preprint arXiv:2401.08092*, 2024.

[128] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[129] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: the power of File-Injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[130] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI (USENIX Symposium of Networked Systems Design and Implementation)*, 2017.

[131] Mingxun Zhou, Elaine Shi, and Giulia Fanti. Pacmann: Efficient private approximate nearest neighbor search. *Cryptology ePrint Archive*, 2024.

[132] Qian Zhou, Hua Dai, Yuanlong Liu, Geng Yang, Xun Yi, and Zheng Hu. A novel semantic-aware search scheme based on bci-tree index over encrypted cloud data. *World Wide Web*, 2023.

[133] Jingchen Zhu, Guangyu Sun, Xian Zhang, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork path: Batching oram requests to remove redundant memory accesses. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

A Layer-wise Caching

Layer	3	2	1
LAION	-	1.8	3.19
SIFT1M	1.86	2.84	2.98
TripClick	1	1.99	2.78
MS MARCO	1.79	2.89	3.05

Table 6: Average number of search steps in each upper layer.

In HNSW, the upper-layer search uses a dynamic candidate list of size one, making it nontrivial to apply the Speculative Neighbor Prefetch directly. From the server's perspective, the memory footprint for upper-layer searches differs from that of the bottom layer. For security, it is necessary to set a separate bound for search steps in the upper layers when not all upper layers are cached locally.

To determine how many search steps in each layer are sufficient, we perform plaintext HNSW searches on the entire query set and collect the average number of search steps in each upper layer, as reported in Tab. 6. Layer 1 is the second-to-last layer. According to Tab. 6, the upper layers' search patterns are similar across all datasets, with an average of 1 to 3 steps per layer. Based on this observation and in order to reduce round trips, we empirically found that a single search step in the second-to-last layer is sufficient to maintain search quality when we cache all but the last two layers.

B ORAM Stash Size Analysis

In this section, we analyze how Compass's ORAM modification impacts stash size. Specifically, we are interested in two aspects of stash usage: the usage between queries and the peak usage during a query. The former helps us estimate memory usage when the Compass client is idle, and the latter indicates the maximum memory demand when processing a query. In the following analysis, we begin by proving the upper bound of Compass's ORAM stash usage between two queries and then analyzing the maximum stash growth during a query.

Now, we show that the stash usage of Compass after eviction can be bounded by the stash usage of the vanilla Ring ORAM. Assume that Compass's ORAM performs a lazy eviction every A' accesses. Let $\mathbf{s} := (\mathbf{a}, \mathbf{x}, \mathbf{y})$ be a sequence of ORAM accesses of size N, where $\mathbf{a} = (a_i)_{i=1}^N$ is the sequence of block addresses, $\mathbf{x} = (x_i)_{i=1}^N$ is the sequence of paths assigned to each address before the access, and $\mathbf{y} = (y_i)_{i=1}^N$ is the sequence of paths assigned to each address after the access. Without loss of generality, we assume that each a_i is unique, A' is a multiple of A, and N is a multiple of A'.

Theorem 2. *Let* $\mathsf{ORAM}_{\mathsf{Compass}}$ *and* $\mathsf{ORAM}_{\mathsf{Ring}}$ *be two ORAM instances with the same set of parameters (Z, S, A, and L).* $\mathsf{ORAM}_{\mathsf{Compass}}$ *is instantiated using Compass's ORAM protocol with a lazy eviction every A' accesses.* $\mathsf{ORAM}_{\mathsf{Ring}}$ *is instantiated with the vanilla Ring ORAM protocol. Let* $\mathsf{st}(\mathsf{ORAM}, \mathbf{s})$ *represent the stash usage. For a given* $\mathbf{s}$*,*

$$\mathsf{st}(\mathsf{ORAM}_{\mathsf{Compass}}, \mathbf{s}) \leq \mathsf{st}(\mathsf{ORAM}_{\mathsf{Ring}}, \mathbf{s}) \tag{3}$$

Following the definition of ∞-ORAM in Ring ORAM, we define $\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}$ as an ORAM that has the same structure as $\mathsf{ORAM}_{\mathsf{Compass}}$, but with infinite bucket size. Similarly, we define $\mathsf{ORAM}^{\infty}_{\mathsf{Ring}}$. We reuse the algorithm G from Ring ORAM as the post-processing algorithm for both $\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}$ and $\mathsf{ORAM}^{\infty}_{\mathsf{Ring}}$.

Lemma 1. *The stash usage of* $\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}$ *after post-processing shares the same randomness as* $\mathsf{ORAM}_{\mathsf{Compass}}$ *after the same access sequence* $\mathbf{s}$*.*

Proof. We notice that delayed eviction can alternatively be viewed as a vanilla Ring ORAM with an increased eviction rate (larger means less frequent) and multiple paths selected per eviction. Ring ORAM's proof of Lemma 1 can be applied directly here to show that such equivalence also holds between $\mathsf{ORAM}_{\mathsf{Compass}}$ and $\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}$ after post-processing. □

Let $\mathsf{h}(\mathsf{ORAM}^{\infty}, a_i)$ be the height of the bucket at which block a_i is stored in ORAM^{∞} after the execution of $\mathbf{s}$.

Lemma 2. *For any* $a_i \in \mathbf{a}$*, we have*

$$\mathsf{h}(\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}, a_i) \leq \mathsf{h}(\mathsf{ORAM}^{\infty}_{\mathsf{Ring}}, a_i) \tag{4}$$

Proof. We know that in Ring ORAM:

$$\mathsf{h}(\mathsf{ORAM}^{\infty}_{\mathsf{Ring}}, a_i) = \mathsf{LCB}(\{p_i\}, \{y_i\}) \tag{5}$$

, where p_i is path selected in the later eviction operation covering the access of a_i and $\mathsf{LCB}(\cdot,\cdot)$ represents the height of the lowest common bucket of two input path sets. Similarly, in Compass, we have:

$$\mathsf{h}(\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}, a_i) = \mathsf{LCB}(P_i, \{y_i\}) \tag{6}$$

, where P_i is the *set* of paths selected during the lazy eviction covering access of a_i. Since $p_i \in P_i$, the height of the lowest common bucket between P_i and y_i is at most $\mathsf{LCB}(\{p_i\}, \{y_i\})$. Therefore, Lemma 2 holds. □

Similar to Ring ORAM, we define T as a rooted subtree of ORAM^{∞}. A rooted subtree includes the root of the ORAM tree, and if a node belongs to T, all its ancestors must also be in T. We define $X_{Compass}(T)$ as the number of real blocks stored in T in Compass ORAM and $X_{Ring}(T)$ as the number of real blocks stored in T in Ring ORAM.

Lemma 3. *If Lemma 2 holds, we have*

$$\forall T, X_{Compass}(T) \leq X_{Ring}(T) \tag{7}$$

Proof. For a block a_i, let $\hat{\mathsf{b}_i}$ be the bucket in $\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}$ that contains a_i and b_i be the bucket in $\mathsf{ORAM}^{\infty}_{\mathsf{Ring}}$ that contains a_i. From Lemma 2 we know that either $\hat{\mathsf{b}_i}$ and b_i are in the same position or b_i is an ancestor of $\hat{\mathsf{b}_i}$. By the definition of T, we know that for any node in T, its ancestors must be in T. However, descendant nodes are not necessarily included. If a block is in T in $\mathsf{ORAM}^{\infty}_{\mathsf{Ring}}$, it might not be in T in $\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}$. Therefore, the number of real blocks in T in $\mathsf{ORAM}^{\infty}_{\mathsf{Compass}}$ is less than or equal to the number of real blocks in T in $\mathsf{ORAM}^{\infty}_{\mathsf{Ring}}$. □

Proof. Finally, it has been shown in Ring ORAM that the stash size exceeds R if and only if there exists a T where the number of real blocks in T is larger than $c(T)+R$, with $c(T)$ defined as the maximum number of blocks T can hold in a non-∞ ORAM. With Lemma 3, we can conclude that the stash usage in Compass ORAM is bounded by that of Ring ORAM. Therefore, Theorem 2 holds. □

After showing the stash usage bounds between queries, we now estimate the extra stash usage during a query. Each Compass query includes a sequence of batched ReadPath operations, followed by a single EvictPath operation. In each ReadPath operation, at most one real block is read from the server into the stash. After the batched ReadPath operations, the stash contains at most $ef \cdot efn$ new blocks. Unlike Ring ORAM, Compass evicts multiple paths simultaneously instead of one. The number of eviction paths is calculated by dividing the paths accessed during ReadPath operations by the eviction rate A, giving $\lceil \frac{ef \cdot efn}{A} \rceil$ paths. During eviction, the client reads Z blocks for each bucket on the selected paths. After reading selected paths, the stash grows by at most $\lceil \frac{ef \cdot efn}{A} \rceil \cdot Z \cdot L + ef \cdot efn$, where L is the number of levels in the ORAM tree. Following this analysis, the extra stash overhead is approximately 80MB on MS MARCO with the evaluation configuration. The actual overhead is smaller because paths share buckets, and not all buckets are full.

C Security Proof

In this section, we provide a formal treatment of the security of Compass. Our proof strategy is to consider two security games that are the original security game with only one of the adversary winning conditions and prove that an adversary cannot fulfill each condition separately more than the threshold allowed. Proving these two statements suffices to prove our theorem in our setting.

In this proof, as we demonstrated in §B, we assume that the main memory is sufficient for the stash during the execution of the protocol. For simplicity, our proof focuses on Compass's protocol within a single layer.

C.1 Condition 1

We begin by proving that the probability of the adversary winning under condition 1 is less than half plus negligible. Our strategy is to show that what the adversary observes when $b=0$ is computationally indistinguishable from what they observe when $b=1$. In the following proof, we omit the execution trace caused by the EarlyReshuffle operations. This is because the adversary knows exactly how many times a bucket has been accessed and therefore when and where EarlyReshuffle happens is public information.

First, we define what the adversary can observe during the game. During the security game, the challenger executes a sequence of queries:

$$Q_b = (q_{1,b}, q_{2,b}, ..., q_{i,b}, ...) \tag{8}$$

Let $O_{i,b}$ be the sequence of ORAM operations for the query $q_{i,b}$. According to Compass's protocol, each query consists of a sequence of batched access operations and an eviction operation at the end. Let m be the number of batches in a query, and we have:

$$O_{i,b} = \mathsf{op}_{\mathsf{access}}(B_1^{i,b}, s_1^{i,b}), ..., \mathsf{op}_{\mathsf{access}}(B_m^{i,b}, s_m^{i,b}), \mathsf{op}_{\mathsf{evict}}, \tag{9}$$

where $\mathsf{op}_{\mathsf{access}}(B_j^{i,b}, s_j^{i,b})$ represents the jth batch access operation of a set of blocks $B_j^{i,b}$ with padding requirement to size $s_j^{i,b}$, and $\mathsf{op}_{\mathsf{evict}}$ represents the eviction operation. Note that $B_j^{i,b}$ is the set of blocks requested by the search algorithm from the ORAM controller, not the set of blocks read by the controller from the remote server. For each block in $B_j^{i,b}$, the controller first checks the position map to find the path assigned to the block. It then looks at the metadata of the buckets along this path to find the block's offset in each bucket. A total of $\lceil \frac{\sum_{j=1}^{m} s_j^{i,b}}{A} \rceil$ paths are selected for eviction in reverse-lexicographic order. From each bucket on these paths, exactly Z blocks are read, and the non-dummy blocks are stored in the stash. Next, all buckets on these paths are updated with newly evicted blocks from the stash, padded with dummies, shuffled, and written back to the server.

Let $P_j^{i,b}$ be the *list* of paths for each block in $B_j^{i,b}$. This is a list because path assignments between blocks may collide. If $|B_j^{i,b}| < s_j^{i,b}$, $P_j^{i,b}$ includes additional randomly selected paths for padding so that $|P_j^{i,b}| = s_j^{i,b}$. Let $P_{evict}^{i,b}$ be the set of paths selected as eviction targets during $\mathsf{op}_{\mathsf{evict}}$, and $\mathcal{T}_{i,b}$ be the trace (or memory footprint) observed by the adversary for query $q_{i,b}$. We have

$$\begin{aligned} \mathcal{T}_{i,b} = {} & \mathsf{t}_{\mathsf{read}}(P_1^{i,b}), ..., \mathsf{t}_{\mathsf{read}}(P_m^{i,b}), \\ & \mathsf{t}_{\mathsf{evict_read}}(P_{evict}^{i,b}), \mathsf{t}_{\mathsf{evict_write}}(P_{evict}^{i,b}) \end{aligned} \tag{10}$$

Here $\mathsf{t}_{\mathsf{read}}(P_j^{i,b})$ means that for each path in $P_j^{i,b}$, one valid block is read from each bucket on the path. The subtlety here

is that if a bucket is part of multiple paths in $P_j^{i,b}$, it is read multiple times. For example, all paths share the root bucket. The Ring ORAM protocol ensures that different blocks are read from the bucket each time. $\mathsf{t}_{\mathsf{evict_read}}(P_{evict}^{i,b})$ means that for each bucket on paths in $P_{evict}^{i,b}$, exactly Z blocks are read, no matter how many paths this bucket belongs to. Similarly, $\mathsf{t}_{\mathsf{evict_write}}(P_{evict}^{i,b})$ writes $Z+S$ blocks (a new full bucket) to each bucket on the paths in $P_{evict}^{i,b}$.

Having defined what the adversary can observe during the game, we now prove that the following lemma holds for Compass.

Lemma 4. *For any pair of $q_{i,0}$ and $q_{i,1}$, if they are of the same type, Compass ensures the corresponding traces $\mathcal{T}_{i,0}$ and $\mathcal{T}_{i,1}$ are of **the same structure**.*

Before we prove this lemma, we first introduce the definition of **the same structure**.

Definition 1. *Two traces are structurally the same if they have*

- *the same number of operations*
- *the same type of operation at the same positions*
- *the same number of paths operated by the operation at the same positions*
- *the same number of blocks read from each bucket*

For example, consider a binary tree with 8 paths, which is labeled from 0 to 7, and the following 5 example traces:

$$\begin{aligned}
&\mathcal{T}_1 : \mathsf{t}_{\mathsf{read}}([1,2]), \mathsf{t}_{\mathsf{evict_read}}(\{3,4\}), \mathsf{t}_{\mathsf{evict_write}}(\{3,4\})\\
&\mathcal{T}_2 : \mathsf{t}_{\mathsf{read}}([1,2]), \mathsf{t}_{\mathsf{evict_read}}(\{3,4\})\\
&\mathcal{T}_3 : \mathsf{t}_{\mathsf{read}}([1,2]), \mathsf{t}_{\mathsf{evict_write}}(\{3,4\}), \mathsf{t}_{\mathsf{evict_write}}(\{3,4\})\\
&\mathcal{T}_4 : \mathsf{t}_{\mathsf{read}}([1,2,3]), \mathsf{t}_{\mathsf{evict_read}}(\{3,4\}), \mathsf{t}_{\mathsf{evict_write}}(\{3,4\})\\
&\mathcal{T}_5 : \mathsf{t}_{\mathsf{read}}([3,4]), \mathsf{t}_{\mathsf{evict_read}}(\{5,6\}), \mathsf{t}_{\mathsf{evict_write}}(\{5,6\})
\end{aligned} \tag{11}$$

If we take $\mathcal{T}_1$ as a reference, among $\mathcal{T}_2$ to $\mathcal{T}_5$, only $\mathcal{T}_5$ is considered to have the same structure as $\mathcal{T}_1$. $\mathcal{T}_2$ has a different length compared to $\mathcal{T}_1$. The second operation in $\mathcal{T}_3$ is a $\mathsf{t}_{\mathsf{evict_write}}$ while the second operation in $\mathcal{T}_1$ is a $\mathsf{t}_{\mathsf{evict_read}}$. $\mathcal{T}_4$ has the same length and operation sequence as $\mathcal{T}_1$. However, the first operation in $\mathcal{T}_4$ operates on 3 paths while the first operation in $\mathcal{T}_1$ operates on 2 paths.

Proof. Now we start the proof for Lemma 4. Take a search query as an example. As shown in Algorithm 1, a search consists of n iterations (line 5) and each iteration contains one batch access (line 14). Therefore, for each search request, the number of operations in the trace is $n+2$, with the extra 2 coming from the eviction. Since n is initialized with $\lceil ef/efspec \rceil$, the number of operations in the trace is fixed for a given set of parameters param. The trace always starts with n $\mathsf{t}_{\mathsf{read}}$ at the beginning and ends with one $\mathsf{t}_{\mathsf{evict_read}}$ and one $\mathsf{t}_{\mathsf{evict_write}}$. Next, we need to show that the number of paths accessed is the same for the same position. For each batch access, the number of requested blocks is padded to a fixed size of t (line 12 - 14), defined as $efspec \cdot efn$. Similarly, the final eviction always operates on $\lceil \frac{n \cdot t}{A} \rceil$ paths. Finally, we show that the same number of blocks is read from each bucket, which follows directly from the Ring ORAM's construction. Therefore, Lemma 4 holds. □

After knowing that $\mathcal{T}_{i,0}$ and $\mathcal{T}_{i,1}$ are structurally the same, we need to show the paths involved during a query, and the blocks we read from each bucket do not reveal any information. This comes directly from the original Ring ORAM construction. The path selected for $\mathsf{t}_{\mathsf{read}}$ appears random to the adversary because the path assignments are chosen uniformly at random. Eviction paths follow a fixed, reverse-lexicographic order, which is both public and static. Each bucket is randomly shuffled before being written back to the server, and any block that has been read becomes invalid until the next reshuffle. Therefore the adversary learns nothing from the position of blocks we read from each bucket.

The final step in our proof here is to invoke the security guarantees of the IND-CCA2 encryption scheme. For each pair of requests $q_{i,0}$ and $q_{i,1}$, the corresponding traces, $\mathcal{T}_{i,0}$ and $\mathcal{T}_{i,1}$, are computationally indistinguishable to the adversary. By the definition of our security games, if the challenger does not abort, Q_0 and Q_1 must have the same length and identical query types at corresponding positions. As a result, the execution trace of Q_0 is computationally indistinguishable from the execution trace of Q_1. Therefore, the adversary's probability of winning condition 1 is no better than random guessing, namely less than half plus a negligible.

C.2 Condition 2

We then show that the probability of the adversary winning in condition 2 is negligible.

Lemma 5. *Assuming that the underlying hash function $\mathcal{H}$ is collision-resistant, then the probability of the adversary winning through condition 2 is negligible.*

Proof. In Compass's protocol, the server operates as remote storage, either sending the client blocks at requested locations or updating its local storage based on the client's messages. Excluding the scenario where the server becomes unresponsive, the only way the server can deviate from the protocol is by sending the client an altered bucket's content. To ensure the integrity and freshness of the buckets that the client reads from the server, Compass uses a Merkle Tree built from collision-resistant hash functions. By the definition of collision-resistance hash functions, the probability of finding two different inputs that produce the same hash output is negligible. It is already well-proven by prior works that a Merkle Tree built with such hash functions ensures data integrity and the probability for an adversary to change the data in a way that the root hash remains the same is negligible.

Although Compass uses an HNSW graph as the underlying data structure rather than a tree, Compass ensures that each node's data, including the coordinates and neighbor list, is stored inside one of the buckets that form a binary tree on the server side. Therefore, the properties of the Merkle hash tree apply directly, which means that the probability of the server successfully forging a response that passes the client's integrity check is negligible. □





Weave: Efficient and Expressive Oblivious Analytics at Scale

Mahdi Soleimani
Yale University

Grace Jia
Yale University

Anurag Khandelwal
Yale University

Abstract

Many distributed analytics applications offloaded to the cloud operate on sensitive data. Even when the computations for such analytics workloads are confined to trusted hardware enclaves, and all stored data and network communications are encrypted, several studies have shown that they are still vulnerable to access pattern attacks. Prior efforts to prevent access pattern leakage often incur network and compute overheads that are logarithmic in dataset size while also limiting the functionality of supported analytics jobs.

We present Weave, an efficient, expressive, and secure analytics platform that scales to large datasets. Weave employs a combination of noise injection and hardware memory isolation to reduce the network and compute overheads for oblivious analytics to a constant factor. Weave also employs several optimizations and extensions that exploit dataset and workload-specific properties to ensure performance at scale without compromising functionality. Weave reduces the end-to-end execution time for a wide range of analytics jobs on large real-world datasets by 4–10× compared to prior state-of-the-art while providing strong obliviousness guarantees.

1 Introduction

Public cloud platforms are increasingly used for storing and processing large volumes of data through distributed analytics frameworks [1–5]. MapReduce (MR) frameworks [1–3], in particular, ensure flexibility, efficiency, and scalability for analytics workloads but raise security concerns since they require trusting the cloud provider with sensitive data.

Ensuring data confidentiality requires protecting the data at rest (*e.g.*, on persistent storage), in flight over the network, and during processing. Early solutions [6, 7] addressed the first two requirements via authenticated encryption, and the third with trusted hardware enclaves (*e.g.*, Intel SGX [8], ARM TrustZone [9] and AMD SEV-SNP [10]). However, recent works [11, 12] have shown that such systems can still reveal sensitive information via *memory and network access pattern leakage*. Specifically, during the execution of map-reduce jobs, the volume of network traffic between pairs of workers (i.e., mappers and reducers) and the memory access patterns at each worker depends on the distribution of data items in the input dataset. An honest-but-curious adversary observing such access patterns can often identify individual data items even when all traffic is encrypted. As such, confidentiality requires secure MR frameworks to ensure these access patterns are *data oblivious*.

While data confidentiality is important for applications operating on sensitive data, security guarantees should not come at the cost of the expressiveness of MR frameworks or their performance at scale. We identify three main goals for secure MR frameworks:

- Strong security, including data confidentiality and protection against access pattern attacks, i.e., *obliviousness*.
- Minimal performance overheads compared to an insecure execution, especially at scale, i.e., for larger dataset sizes and with many distributed workers.
- Little to no restrictions on functionality, i.e., on the types of supported Map and Reduce functions.

We introduce a novel security definition to capture access pattern attacks on distributed analytics platforms by an honest-but-curious adversary in the cloud. The definition, named Indistinguishability under Chosen Dataset and Job Attack (or IND-CDJA, §2.2), informally requires that the distribution of network communications between workers and memory accesses inside workers observed during the execution of a secure MR framework is independent of the distribution of data items in the underlying input dataset. While this security requirement introduces a (provably) hard trade-off between arbitrary MR functionality and bounded performance overheads achievable by *any scheme* (§2.3), a practical assumption on the intermediate data generated by MR jobs (similar to prior works in this space [11, 12]) permits bypassing this tradeoff to achieve bounded overheads.

Unfortunately, existing approaches for mitigating access

pattern leakage in MapReduce frameworks either limit functionality beyond the assumptions required by IND-CDJA, incur high performance overheads, or both, to achieve security (§2.4). These approaches fall into two categories — sort-based (*e.g.*, Opaque [12]) and load-balancing (*e.g.*, Shuffle & Balance [11]). While those in the former category rely on cryptographic schemes like oblivious sort [13], those in the latter rely on oblivious shuffle [14]. Unfortunately, both schemes have well-known log-linear complexity, which translates to significant performance overheads in practice — often increasing the end-to-end job execution times by an order of magnitude or more than an insecure baseline (§5). Moreover, both approaches limit functionality beyond what is required to achieve obliviousness: the former on non-associative operations [12], and the latter on sort-based analytics [11].

We present Weave, an oblivious MapReduce framework that meets all three goals for secure analytics. Weave uses the load balancing approach as a starting point for network obviousness, but leverages principles of *noise-injection* explored in the context of oblivious storage systems [15–19] to achieve *constant* factor ($\sim 3\times$ or lower) overheads. Unlike prior approaches that employ data-agnostic shuffle mechanisms, our performance gains stem from Weave exploiting the distribution of data in the input dataset itself (§3). Specifically, Weave's shuffle phase estimates this distribution during execution and uses it to inject just enough "fake" network traffic so that the observable access pattern between pairs of workers is independent of the underlying distribution, i.e., network communications are *data oblivious*. Indeed, we show that Weave achieves the lowest network overhead achievable by any noise-injection scheme for oblivious communications in MapReduce frameworks. We also show that while the data structures employed by our noise injection mechanism can introduce memory access pattern leakage, judicious use of hardware-protected Enclave Page Cache (EPC) can avoid them with little to no overhead. We develop further optimizations and extensions to Weave that exploit dataset and workload-specific properties to improve its performance at scale and ensure rich functionality.

We show that Weave is secure under IND-CDJA, achieving strong security guarantees with obliviousness even with constant performance overheads (§4). We implement Weave atop Apache Spark [2] and evaluate it on several real-world workloads. Weave's execution times for these workloads is 4–10$\times$ lower than state-of-the-art systems with comparable security guarantees. Moreover, its performance scales linearly with dataset size and the number of workers (§5). The code for Weave, the datasets, and workloads used in our evaluation are available at `https://github.com/yale-nova/weave`.

2 Background and Motivation

We begin by outlining the secure MapReduce execution and threat model and demonstrating how sensitive information is leaked via network and memory access patterns even when the data is encrypted. We then discuss shortcomings of prior strategies in navigating the fundamental performance-functionality tradeoff in secure MapReduce execution.

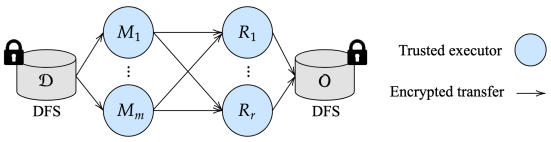


Fig. 1: **System model for secure MapReduce execution.** Tasks run in TEEs while data is encrypted in flight and at rest.

2.1 Secure Analytics Execution

We consider the secure execution of analytics jobs on a cloud-hosted MapReduce (MR) platform [1–3], as depicted in Figure 1. The MR platform reads and writes data from a cloud-hosted Distributed File System (DFS) [20, 21]. The client uploads an encrypted dataset D containing n equally-sized key-value pairs[1] to the DFS and submits a job to be executed on D using the MR framework. The MR framework runs the computations required for the job in secure enclaves — trusted containers of code and data isolated from the rest of the system — to ensure data confidentiality [6, 11, 12]. Data is only decrypted within the enclaves and re-encrypted before network communications and other I/O operations.

MR Job execution. An MR job is specified using a pair of map and reduce functions $(\mathsf{Map}, \mathsf{Reduce})$. Map takes a key-value pair as input and outputs a list of intermediate key-value pairs. Reduce takes as input an intermediate key and a set of values for that key. It combines these values using user-specified logic to output a new set of values.

MR frameworks comprise a centralized controller and multiple distributed workers. The controller receives job requests and schedules their execution across the workers in three sequential phases: map, shuffle, and reduce. In the map phase, the input dataset is divided into m splits, and a distinct worker ("mapper") processes each split in parallel. Each mapper executes Map on every key-value pair in their split to generate the intermediate key-value pairs. Next, the shuffle phase regroups mapper-generated intermediate key-value pairs into r partitions based on the intermediate key. The partitioning function (*e.g.*, hash or range-based) ensures that intermediate values with the same key are grouped into the same partition. Each partition is then processed in parallel by a distinct worker ("reducer") in the reduce phase, where each reducer executes Reduce on groups of values with the same intermediate key.

Threat Model. Similar to prior work on secure MR execution [6, 11, 12], we consider an honest-but-curious adversary that monitors the data at rest, the accesses to encrypted

[1] While plaintext datasets can have variably-sized records, we assume encryption pads them to the same size to avoid leaking record sizes [12].

memory pages inside MR workers, and the network communications between MR workers — the cloud provider is a typical example of such an adversary. We next detail the key components and assumptions of our threat model:

- **Computations.** All computations (*e.g.*, Map, Reduce executions) are assumed to run within Trusted Execution Environments (TEEs). TEEs provide attestation mechanisms so the adversary cannot deviate from the MR protocol. Moreover, the adversary cannot observe computations performed within the TEE.
- **Memory access patterns.** While memory access patterns within the TEE are generally visible to an adversary, each TEE is equipped with a small amount of Enclave Page Cache (EPC) [22] which ensures memory obliviousness, i.e., hides access patterns to data placed within them from the adversary. We defer the details of EPC security guarantees and their practical implementation to §3.3 and §3.6.
- **Network access patterns.** The adversary can observe worker communications (i.e., the size of traffic communicated between worker pairs) throughout a job's execution.
- **Data and code.** While the data handled by the MR framework is always encrypted at rest and in flight, the adversary can observe the size of the input dataset, the code for the functions Map and Reduce, and the allocation of datasets to workers. Polynomial-time computations can be performed on the above data to extract information about the underlying dataset.
- **Other side-channels.** Weave assumes enforced attestation and access control (*e.g.*, Azure CVMs [23, 24]), which constrain the attack surface and make sophisticated attacks (*e.g.*, hardware side-channel or physical access attacks) infeasible in practical cloud deployments [25]. Time and per record size difference channel attacks are out of Weave's scope. We discuss these channels further in §6.

Although secure analytics execution outlined above ensures that data is never stored or processed in plaintext during job execution, prior work [11, 12] has shown that worker communication patterns can still reveal sensitive data to an honest-but-curious adversary, as we detail next.

2.2 Access Pattern Leakages

Even in secure MapReduce frameworks, the communication patterns between workers for a given job can be related to the contents of the encrypted input, which an adversary can leverage to learn sensitive information about the input. We characterize this leakage using a simple MR job on chronologically ordered patient datathat counts cases per disease:

Map(patientID, (disease, date)):	Reduce(disease, $\{c_1, c_2, \cdots, c_j\}$):
Emit (disease, 1)	Emit (disease, $\sum c_i$)

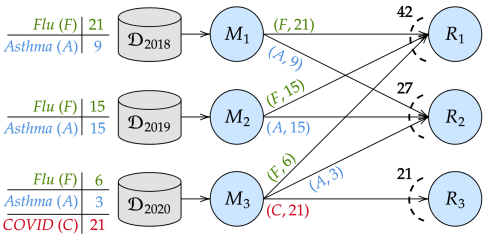


Fig. 2: **Example detailing mapper-to-reducer communication volume for an analytics job execution over medical records (§2.2).** The execution suffers from both split- and distribution-based leakage.

In the above job, Map extracts the disease and case count per record, and Reduce aggregates all case counts for a disease.

Figure 2 shows the access pattern leakage when executing this query on a medical dataset of patient records. We assume that the adversary knows the nature of the input: chronologically ordered medical records from 2019–2021, and wants to correctly label the communicated records between specific mapper-reducer pairs with the associated disease.

Split-based leakage. The set of intermediate key-value pairs generated by different mappers is often tied to how the key-value pairs may have been divided across input splits of the dataset. Since mappers split the chronologically ordered data, only some of the mappers' (M_3 in Figure 2) splits contain COVID-19 records, as cases were only diagnosed after 2019. While the adversary may not know which mappers will process COVID-19 cases, it knows that only a few mappers will process them. Since MapReduce semantics require all COVID-19 records to be processed on the same reducer, the adversary can identify which records correspond to COVID-19 cases based on the distinct communication pattern between mappers and reducers. Specifically, by noting that reducer R_3 receives records only from mapper M_3, even when all communications are encrypted, an observer can deduce that R_3 is most likely processing COVID-19 records.

Distribution-based leakage. The distribution of key-value pairs across reducers is tied to the distribution of intermediate keys generated during the map phase. If the adversary in Figure 2 knows the relative frequency of disease cases, then it can identify patients' diseases based on the number of records received by each reducer, *e.g.*, knowing Flu is $\sim 1.5\times$ more frequent than Asthma, the adversary can deduce that records sent to R_1 are for Flu, and those sent to R_2 are for Asthma.

In both examples, the adversary leverages public knowledge about the dataset to infer information about records sent between mapper-reducer pairs. While our examples are simplified for elucidation, prior work has shown that such attacks, combined with other side channels, can accurately reconstruct sensitive fields like age, marital status, and birthplace for individuals in US census data [11].

IND-CDJA. To capture the security of a MR framework

against such access pattern leakage-based attacks, we introduce a formal notion of security: Indistinguishability under Chosen Dataset and Job Attack or IND-CDJA. While we defer formal security definitions to §4, informally, a system is secure under IND-CDJA if the adversary cannot distinguish between the execution of a MR job on two different datasets of the same input size. In the context of access pattern leakage, the adversary cannot correlate the observed patterns of memory access inside workers and network communications among workers during execution to the distribution of data in the underlying private input dataset, i.e., an IND-CDJA-secure system ensures that the data distribution in the input (private) is independent of the access patterns (observable).

2.3 Performance-Functionality Tradeoff

A secure MR framework should support three main properties (§1): (i) *security* under IND-CDJA, (ii) *minimal performance overheads* relative to an insecure execution, and, (iii) minimal restrictions on *functionality*, i.e., the jobs that the framework can support. Unfortunately, IND-CDJA security exposes a hard tradeoff between functionality and the performance overheads that must be incurred to achieve this security:

Theorem 2.1 *No* IND-CDJA*-secure scheme can support arbitrary* MapReduce *jobs with bounded bandwidth overheads.*

Proof. Consider a MapReduce job, where the map function emits C intermediate key-value pairs ($C > 1$) if it encounters a particular key k^* and outputs one key-value pair otherwise. It is easy for an adversary to distinguish between two executions of the job, one on a dataset D^* that contains k^* and the other on a dataset D that does not (note, $|D^*| = |D| = n$). Specifically, while D^* causes $C+n-1$ intermediate key-value pairs to be exchanged between mappers and reducers, D results in n key-value pairs being exchanged.

To make the two executions indistinguishable, an IND-CDJA-secure must ensure both executions exchange *at least* $C+n-1$ records; exchanging fewer records will cause the output of the first execution (on D^*) to be incorrect. In other words, the bandwidth overhead of an IND-CDJA-secure scheme is *at least* $\frac{C+n-1}{n}$. However, since C can be arbitrarily large (*i.e.*, $C \to \infty$), the bandwidth overhead of an IND-CDJA-secure scheme will also be unbounded. ■

We note that the unbounded overhead in our proof's counterexample stems from the map function's ability to generate *arbitrary* number of intermediate key-value pairs. If the number of intermediate key-value pairs generated by a map function, c, is upper-bounded to a fixed constant C, then an IND-CDJA-secure scheme could achieve bounded performance overheads. Indeed, prior works in this space [11, 12] implicitly assume a bound C and achieve security that is semantically similar to IND-CDJA[2]. While we will discuss how prior approaches support $c = 1$ soon, we summarize their approach to supporting $c \neq 1$ below.

$c < 1$. These are common jobs that *filter* a subset of the dataset using some predicate. A simple approach for supporting such operations in an IND-CDJA-secure scheme is for Map to output exactly one intermediate key-value pair for every input record but append a 'valid' bit to the output. This approach requires the Reduce function to ignore invalid key-value pairs but always ensures $\hat{n} = n$ and allows IND-CDJA-secure to support this class with the same property as the $c = 1$ case.

$c > 1$. Theorem 2.1's proof leveraged a Map function in this class. With an upper-bound C on c, a secure scheme can add "filler" key-value pairs to the Map outputs (with invalid bits) to ensure exactly C intermediate key-value pairs per input record. Again, the Reduce function filters invalid key-value pairs. An adaptation of IND-CDJA-scheme for the $c > 1$ case can achieve bounded bandwidth, albeit with an additional $\frac{C}{c_{avg}}\times$ overhead, where c_{avg} is the average Map output size.

We provide a detailed analysis of various operations supported by MapReduce schemes, SQL frameworks built atop them, and typical MapReduce patterns that use them, along with the value of c they use in the full version of our paper [26]. Our key takeaway is that commonly used operations and application patterns employ ($c \leq 1$), which can be supported with low overheads (§3). Since the approach for supporting all of the above cases reduces to an IND-CDJA-secure scheme that can support $c = 1$, we restrict our focus to the $c = 1$ case (with $\hat{n} = n$) for the rest of the paper, unless explicitly mentioned.

2.4 Prior Approaches

Prior schemes for mitigating access pattern leakage in MR frameworks employ security semantically similar to IND-CDJA, and are either sort- or load-balancing-based.

Sort-based schemes use oblivious sort algorithms [13, 27] for intermediate key-value pairs exchanged between map and reduce phases. As a result, each reducer receives an equally sized contiguous segment of the sorted intermediate key-value pairs. Unfortunately, practical realizations of oblivious sort in MR frameworks suffer from two main shortcomings. First, they incur high compute and network overheads. Opaque [12], a representative sort-based approach, employs column-sort [13], which requires several rounds of processing and network shuffles, often resulting in long execution times (§5). Second, the use of oblivious sort makes the support for non-associative Reduce (*e.g.*, median) non-trivial. As such, systems like Opaque only support associative Reduce (*e.g.*, count), limiting functionality beyond the minimum restrictions for IND-CDJA-secure approaches.

Load-balancing schemes prevent distribution-based leakage by assigning equal numbers of intermediate key-value pairs to reducers. To this end, intermediate key-value pairs

[2] Details on differences in security guarantees are deferred to §4.

generated by mappers are first bin-packed using a first-fit decreasing approach [28] across reducers (with random assignment of bins to reducers). Then, all bins are padded to the same size, which depends on the frequency of the most popular intermediate key. To avoid split-based leakage, these approaches require the dataset to be obliviously shuffled before the map phase, *e.g.*, Shuffle & Balance [11], a representative load-balancing approach, uses Melbourne shuffle [14]. Unfortunately, these schemes also observe two shortcomings. First, while bin-packing is more efficient than oblivious sort (both in theory and in practice), oblivious shuffle observes log-linear network complexity [14, 29] and still presents a performance bottleneck (§5). Second, the bin-packing approach assigns intermediate key-value pairs to reducers based on key frequencies, obviating support for sort-based or any user-defined partitioning function — again, imposing more restrictions on functionality than an IND-CDJA-secure scheme must.

Other approaches include differential obliviousness [30, 31] and secure multi-party computations [32]. Although the former approach incurs overheads comparable to Weave, it employs significantly weaker security and remains susceptible to practical attacks like Figure 2's example. The latter approach, unfortunately, incurs impractically high performance overheads, making them unsuitable for MapReduce frameworks. We do not consider these approaches in our work.

3 Weave Approach

Weave builds on the load-balancing approach and leverages principles of noise-injection employed in oblivious storage [15–18] to improve its performance and reduce its restrictions on MapReduce functionality. Weave decomposes analytics job execution into two parts:

Initialization. Weave first initializes worker nodes to be used as mappers, reducers, and for additional required processing to prevent access pattern leakage. It then generates secret keys for encryption and the pseudorandom generator (PRG), sharing them among the workers via secure channels. These are stored in each worker's secure enclave during the secure analytics job execution.

Job execution. Similar to secure analytics execution on prior MR frameworks [6, 11, 12] (§2.1), Weave's job execution still comprises the map and reduce phases, where workers run the Map and Reduce functions within TEEs with secure enclaves and memory oblivious accesses inside EPC sections. However, since the key source of access pattern leakage stems from an observer analyzing the communication volumes between mapper-reducer pairs (§2.2) during the shuffle phases, Weave replaces this phase with three new phases that prevent such leakage, namely the *random-shuffle*, *histogram*, and *balanced-shuffle* phases (Figure 3(a)). While random-shuffle only involves network communications, the histogram and balanced-shuffle phases involve both computations performed by workers and network communications. We refer to the w workers in these phases as *weavers* (W_i).

At a high level, random-shuffle (①) prevents any split-based leakage by randomly distributing intermediate key-value pairs generated by mappers to weavers in the next phase, effectively making the volume of network traffic between mapper-weaver pairs independent of how the data was distributed across splits (§3.1). The histogram (②) and balanced-shuffle (③) phases, on the other hand, collectively prevent distribution-based leakage by making the volume of network traffic between weaver-reducer pairs independent of the distribution of intermediate keys generated during the map phase (§3.2). Finally, while these phases ensure no access pattern information is revealed via network communications between workers, any leakage due to the memory accesses performed within the TEEs during these phases is prevented by placing the associated state within the Enclave Page Cache (EPC), an isolated memory region inside the TEE (§3.3). The following subsections describe these three phases in detail.

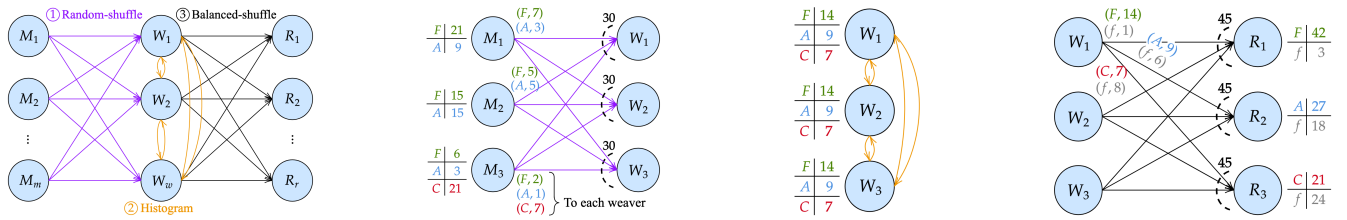
3.1 Preventing Split-based Leakage

As noted in §2.2, split-based leakage stems from intermediate key-value pairs generated by different mappers being tied to how input records may have been divided across input splits of the dataset. To prevent such leakage, Weave's random-shuffle phase requires each mapper M_i to route each intermediate key-value pair to a pseudorandomly chosen weaver W_j. Since the choice of the weaver processing a key-value pair is independent of how records were divided across the input dataset splits, Weave avoids split-based leakage (formal proof in §4).

An important consequence of such a shuffle is that each weaver receives a *proportionate* distribution of intermediate keys. In particular, if the map phase generates a total of $\hat{n}$ intermediate key-value pairs and $\hat{n}_k$ intermediate key-value pairs with key $\hat{k}$, then random shuffle ensures that each weaver receives, in expectation, $\frac{\hat{n}_{\hat{k}}}{w}$ key-value pairs with key $\hat{k}$. Figure 3(b) illustrates this on the simple example from §2.2.

3.2 Preventing Distribution-based Leakage

Distribution-based leakage, on the other hand, stems from the number of key-value pairs received by each reducer being tied to the distribution of intermediate keys generated during the map phase. Our approach for achieving both goals draws inspiration from prior literature on noise-injection [15–18] for oblivious storage, which employs principled use of fake queries to a storage system to make the observable access pattern across data items indistinguishable from a uniform random one. The histogram and balanced-shuffle phases in Weave realize a form of noise injection where the weavers leverage the histogram of intermediate keys to transfer fake and real intermediate key-value pairs to reducers. This allows Weave to ensure the volume of network traffic between



(a) Weave Overview (b) Random-shuffle phase (c) Histogram phase (d) Balanced-shuffle phase

Fig. 3: **Secure analytics execution in Weave.** (a) Weave replaces the shuffle phase with three new phases: ① random-shuffle, ② histogram and ③ balanced-shuffle (§3). (b-d) demonstrates Weave's execution on the example from Figure 2. (b) Random-shuffle distributes intermediate key-value pairs from mappers randomly across weavers: in our example, each weaver receives the same number of Flu, Asthma, and COVID-19 records. (c) Each weaver broadcasts a histogram of its received intermediate key-value pairs, informing all weavers of the global histogram of Flu, Asthma, and COVID-19 records (d) Balanced-shuffle assigns Flu records to reducer R_1, asthma to R_2, and COVID-19 to R_3. To ensure that each reducer receives $\texttt{kv_tot} = 45$ key-value pairs ($\alpha = 1.5$), each weaver sends fake key-value pairs (key f) to each reducer.

weaver-reducer pairs is roughly equal and independent of the distribution of intermediate keys. We detail these phases next.

Histogram phase. This phase aims to construct a global histogram of intermediate keys generated by each mapper at every weaver without leaking any information about the distribution of keys in the dataset.

To this end, each weaver W_i first creates a local histogram h_i over the $\hat{n}_i$ intermediate keys it receives from mappers in the random-shuffle phase. Since the random-shuffle phase ensures that $\hat{n}_i$ is independent of the underlying data distribution in the input dataset, the histogram h_i is padded to $\hat{n}_i$ entries to prevent any leakage based on histogram sizes. The padded histogram is then encrypted and broadcasted to all other weavers $W_{j\neq i}$. As such, weaver W_i also receives histograms h_j from all other weavers $W_{j\neq i}$, which it combines to construct a global histogram $\hat{\mathsf{h}}$ over all intermediate keys. By the end of this phase, every weaver has the same global histogram $\hat{\mathsf{h}}$. Figure 3(c) illustrates the histogram phase by continuing the example from Figure 3(b).

Balanced-shuffle phase. Having computed the global histogram, the weavers ultimately need to decide on the destination reducer (among all r reducers) for each intermediate key-value pair while satisfying two constraints. First, correctness requires all key-value pairs with the same key to be sent to the same reducer. Second, obliviousness requires the number of key-value pairs sent between each weaver-reducer pair to be independent of the data distribution in the input dataset.

The balanced-shuffle phase in Weave employs a form of noise injection; Algorithm 1 details this approach. Weave first fixes the number of intermediate key-value pairs (`kv_tot`) any reducer receives to a value independent of the intermediate key distribution. We set `kv_tot` to $\alpha \cdot \frac{\hat{n}}{r}$, where $\alpha > 1$ is independent of the key distribution. Weave then sends a combination of real and fake key-value pairs from the weavers to meet the fixed quota per reducer.

Algorithm 1 BalancedShuffle($\hat{\mathsf{h}}$)

At each weaver W_i:

1: $\texttt{kv_tot} \leftarrow \frac{\alpha \cdot \hat{n}}{r}$
2: $k \leftarrow$ first intermediate key in $\hat{\mathsf{h}}$
3: **for** $j = 1$ to r **do**
4: $\quad \texttt{kv_real}[j] \leftarrow 0$
5: $\quad$ **while** $\texttt{kv_real}[j] + \hat{\mathsf{h}}[k] < \texttt{kv_tot}$ **do**
6: $\quad\quad$ Assign all KV pairs for key k at W_i to R_j
7: $\quad\quad \texttt{kv_real}[j] \leftarrow \texttt{kv_real}[j] + \hat{\mathsf{h}}[k]$
8: $\quad\quad k \leftarrow$ next intermediate key in $\hat{\mathsf{h}}$
9: **for** $j = 1$ to r **do**
10: $\quad \texttt{kv_fake}[j] \leftarrow \texttt{kv_tot} - \texttt{kv_real}[j]$
11: $\quad$ **for each** $l = 1$ to $\texttt{kv_fake}[j]$ **do**
12: $\quad\quad d \leftarrow$ Roll a w-sided unbiased die
13: $\quad\quad$ **if** $d = i$ **then**
14: $\quad\quad\quad$ Assign a fake KV pair from W_i to R_j
15: Send all fake+real KV pairs from W_i to their assigned reducers

Assigning real key-value pairs: Weave first leverages the global histogram $\hat{\mathsf{h}}$ to greedily assign real key-value pairs with the same key across as few reducers as possible while ensuring (i) each reducer receives no more than `kv_tot` real key-value pairs, and (ii) intermediate key-value pairs with the same key are assigned to the same reducer (lines 2–8). Note that since all weavers have a roughly equal proportion of key-value pairs for each intermediate key ($\frac{\hat{n}_i}{w}$ key-value pairs with key k_i in expectation) due to the random shuffle phase (§3.1), each reducer R_j receives, in expectation, $\frac{\texttt{kv_real}[j]}{w}$ of the total $\texttt{kv_real}[j]$ real key-value pairs from each weaver.

Assigning fake key-value pairs: After assigning real key-value pairs, the remaining space in each reducer's fixed-quota (`kv_tot`) is filled using randomly generated fake key-value pairs (lines 9–14). Hiding distribution-based leakage requires that roughly the same number of intermediate key-value pairs are sent between each weaver-reducer pair. Since the proportion of real key-value pairs sent from various weavers to each reducer is already roughly equal, all that remains is to ensure that the proportion of fake key-value pairs sent from various weavers to a reducer is also roughly equal. This is

realized via the Bernoulli trials in lines 12–14 — a w-sided die is cast for each fake key-value pair, and the value of the roll decides which weaver gets to send that fake key-value pair. This ensures that each of the weavers generates, in expectation, $\frac{\texttt{kv_fake}[j]}{w}$ of the total $\texttt{kv_fake}[j]$ fixed number of fake key-value pairs received by reducer R_j. Combined with the real key-value pairs, each weaver sends, in expectation, $\frac{\texttt{kv_fake}[j]+\texttt{kv_real}[j]}{w} = \frac{\texttt{kv_tot}}{w}$ key-value pairs to each reducer R_j, ensuring obliviousness. Note that each reducer R_j receives exactly $\texttt{kv_tot}$ records.

A subtle issue in the above approach is that all weavers must agree upon the die roll values — otherwise, multiple weavers may send the same fake key-value pair at the same time, or no weaver may send it. One approach is to perform all die rolls at a centralized entity (*e.g.*, the centralized controller or a designated weaver) and disseminate their values to all weavers. However, this approach adds a potential scalability bottleneck. Weave opts for an even simpler solution — it distributes the same PRG to each of the weavers during initialization (§3), ensuring that each of the weavers independently generates the same die roll values and avoiding any runtime communication overheads for this purpose.

Finally, each weaver sends the real and fake intermediate key-value pairs to the reducers determined by the bin-packing and noise-injection approaches outlined above. The reducers, in turn, collect the intermediate key-value pairs from the weavers, decrypt them, and drop the fake entries. They then proceed with the reduce phase similar to traditional MR: applying Reduce to groups of intermediate key-value pairs with the same key and adding the results to the output with requisite padding and encryption.

Figure 3(d) continues the example from Figures 3(b) and 3(c) to show how the data records from a weaver are distributed across different reducers in the balanced-shuffle phase to prevent distribution-based leakage.

Choice of α. Clearly, larger values of α incur larger network bandwidth overhead, and therefore, the end-to-end execution time for MR jobs in Weave. While $\alpha > 1$ is a trivial lower bound, correctness mandates a larger α value:

Theorem 3.1 *For any $\alpha < \frac{2r}{r+1}$, there exist $\hat{n}$ intermediate key-value pairs that cannot be assigned into r reducers such that each reducer processes exactly $\frac{\alpha\cdot\hat{n}}{r}$ key-value pairs.*

Proof. Consider $\hat{n}$ intermediate key-value pairs with $r+1$ distinct keys, each with an equal number of intermediate key-value pairs, i.e., $\forall k, \hat{\mathsf{h}}[k] = \frac{\hat{n}}{r+1}$. If $\alpha < \frac{2r}{r+1}$, each reducer processes exactly $\texttt{kv_tot} = \alpha \times \frac{\hat{n}}{r} < \frac{2\hat{n}}{r+1}$ key-value pairs.

A single reducer must process all intermediate key-value pairs with the same key for correctness. As such, no reducer can be assigned intermediate key-value pairs for two or more distinct keys since $2 \times \frac{\hat{n}}{r+1} > \texttt{kv_tot}$. Accommodating all $r+1$ keys requires at least $r+1$ reducers, a contradiction. ■

Fortunately, $\alpha = \frac{2r}{r+1}$ is achievable, and can indeed be achieved by Algorithm 1 as we show next:

Theorem 3.2 *If $\alpha \geq \frac{2r}{r+1}$, $\hat{n}$ intermediate key-value pairs can be always assigned to r reducers such that each reducer processes exactly $\frac{\alpha\cdot\hat{n}}{r}$ key-value pairs, provided $\max_k \hat{\mathsf{h}}(k) \leq \frac{\alpha\cdot\hat{n}}{r}$.*

Proof. Consider a modified Algorithm 1 that permits intermediate key-value pairs for the same key to be split across two reducers in lines 2–8; note that since $\max_k \hat{\mathsf{h}}(k) \leq \frac{\alpha\cdot\hat{n}}{r}$, the intermediate key-value pairs associated with the same key may be split across at most two reducers. This assignment ensures all the key-value pairs will be assigned to $r_1 = \frac{\hat{n}}{\frac{2\hat{n}}{r+1}} = \frac{r+1}{2}$ reducers, regardless of the key distribution.

Let $\mathcal{K}_s$ be the set of intermediate keys whose values are split across two reducers. Note that $|\mathcal{K}_s|$ is at most $r_1 - 1 = \frac{r-1}{2}$. For each key $k \in \mathcal{K}_s$, the modified algorithm then reassigns all of k's key-value pairs to one of the $r - r_1 = \frac{r-1}{2}$ reducers that have not been assigned any keys yet. This is a feasible assignment, completing the proof[3]. ■

Note that $\alpha = \frac{2r}{r+1}$ still ensures that the traffic volume between weaver-reducer pairs is independent of the key distribution. However, as noted in Theorem 3.2's statement, this value of α requires the maximum number of intermediate key-value pairs associated with the same key (i.e., $\max_k \hat{\mathsf{h}}(k)$) must not exceed $\frac{\alpha\cdot\hat{n}}{r}$. Intuitively, this constraint captures Weave's requirement of the maximum intermediate key popularity being less than a single reducer's capacity ($\texttt{kv_tot}$) — a reasonable assumption in any MR job. Indeed, the maximum intermediate key popularity is much smaller than the reducer capacity in practice — no more than $\sim 5\%$ in our evaluations (§5). However, since α (and therefore, $\texttt{kv_tot}$) is a configurable parameter in Weave, it is possible to increase α (i.e., reducer capacity) to support a larger maximum key popularity — we analyze Weave's sensitivity to α in §5.4.

3.3 Avoiding Memory Access Pattern Leakage

While the random-shuffle, histogram, and balanced-shuffle phases prevent distribution-based leakage and split-based leakage, we note that *memory access patterns* during these phases can still leak sensitive information about data distribution. Similar to prior work [8, 10, 33–35], Weave leverages the Enclave Page Cache (EPC) [36, 37], a limited pool of oblivious memory that hides access patterns using a combination of hardware isolation and lightweight side-channel defenses (§3.6). We confine all memory-dependent sensitive states to this region and structure the design to minimize its footprint for scalability.

To facilitate this, Weave classifies memory accesses during the three phases into two classes: *data-independent* and *data-dependent*. Data-independent accesses (*e.g.*, scanning the entire dataset) do not reveal any distribution-dependent information about the dataset and can be kept outside EPC regions

[3] Algorithm 1 can always achieve the same overhead.

without affecting security (§4.1). In contrast, data-dependent accesses must use protected EPC memory:

Random-shuffle. Since random-shuffle scans through and pseudo-randomly assigns key-value pairs to different weavers, it performs no data-dependent memory accesses. As such, Weave does not use EPC memory during this phase.

Histogram. Since the histogram phase aggregates key counts in separate memory locations, the frequency of accesses to each location reveals the distribution of key-value pairs. Thus, each weaver stores its local histogram (h_i) and global histogram ($\hat{h}$) in the EPC.

Balanced-shuffle. In Algorithm 1, the buffers needed for the aggregation of the real and fake key-value pairs for reducer R_j at weaver W_i, and the `kv_real` and `kv_fake` counters that track W_i's real and fake key-value pair counts per reducer have data-dependent accesses and are kept inside EPC memory.

3.4 Weave Optimizations & Extensions

Sampled histogram for scalability. During Weave's histogram phase, each weaver broadcasts a local histogram containing $\sim \lceil \frac{\hat{n}}{w} \rceil$ entries to all other weavers; this presents a non-trivial overhead for large datasets, limiting Weave's scalability. To avoid this overhead, we observe that Weave's security against distribution-based leakage stems from the *indistinguishability* of traffic volume between weaver-reducer pairs. As such, weavers do not require *precise* knowledge of the global histogram $\hat{h}$; an *estimate* of $\hat{h}$ can be used as long as it can still ensure the indistinguishability guarantee. Weave thus builds the global histogram at each weaver using *random samples* of the intermediate key-value pairs. As noted in §3.3, histogram aggregations occur inside EPC; thus, sampling also reduces EPC memory requirement during histogram phase.

Specifically, each weaver only computes a local histogram over $s = \beta \cdot \frac{\hat{n}}{w}$ random samples out of the total $\frac{\hat{n}}{w}$ intermediate key-value pairs it receives, where $\beta < 1$ is the sampling factor. It then pads each local sampled histogram to $\lceil \frac{\beta \cdot \hat{n}}{w} \rceil$ entries and broadcasts it to other weavers. Each weaver aggregates the sampled histograms from other weavers and scales the key counts by $1/\beta$ to get an approximate global histogram.

We note that the approximation for the histogram can introduce variations in the weaver-reducer traffic during the balanced-shuffle phase that an adversary can observe. This requires additional noise (i.e., fake key-value pairs) to be added to weaver-reducer traffic to hide such variations. If we increase the traffic by a factor of $(1+\delta)$, then the Chernoff bound [38] allows us to ensure that the probability ε of successfully distinguishing the traffic volumes is $\varepsilon < \exp(-\frac{\beta\hat{n}\delta^2}{2})$, as captured in the following theorem:

Theorem 3.3 *For any polynomial-time adversary $\mathcal{A}$, if Weave's histogram phase samples $\beta\hat{n}$ intermediate key-value pairs and its balanced-shuffle phase sends $(1+\delta)\times$ more fake key-value pairs for some β and δ, $\mathcal{A}$'s advantage $\mathbf{Adv}^{ind\text{-}cdja}_{Weave}$ is negligible.*

Proof. Let the random variable X_i denote the number of sampled queries received by reducer R_i; since the sampled key-value pairs are each sent to a random reducer, $X_i \sim B(\beta\hat{n}, \rho)$ where ρ is the probability that a sampled intermediate key-value pair is sent to reducer R_i.

Given some small $\delta > 0$, the Chernoff bound [39] shows that the probability of ρ falling outside the bound of $[\frac{1}{r} - \delta, \frac{1}{r} + \delta]$ is an exponentially decreasing function of $\hat{n}, \beta$, and δ. Specifically, we obtain an upper-bound on the probability that our estimate $\frac{\alpha}{r}$ is off from ρ by more than δ if the sample size is $\beta\hat{n}$:

$$\Pr\left[\rho \geq \frac{\alpha(1+\delta)}{\hat{n}}\right] < \exp(-\frac{\hat{n}\beta\delta^2}{2}) \qquad (1)$$

Now, let us modify the balanced-shuffle phase (Algorithm 1) to send $(1+\delta)\times$ more fake intermediate key-value pairs to each reducer. Then, the advantage that a polynomial computation-bounded adversary has in distinguishing between the traffic distributions between weavers and reducers and a uniform random distribution is exponentially small, specifically, the probability of ρ being outside the bound given by Inequality 1.

Therefore, Weave is still IND-CDJA-secure if its histogram phase uses sample size $\beta\hat{n} \in O(\hat{n})$. ■

Our implementation uses a sampling factor β of 1% during the histogram phase and additional noise 5% (δ) during balanced-shuffle while still ensuring negligible ε.

Efficient support for associative reduce. Under our threat model, an adversary is aware of the nature of the analytics job being executed (§2.2). We can, therefore, employ targeted optimizations for associative reduce functions since this reveals no additional information to the adversary.

To this end, we modify the balanced-shuffle algorithm to *eliminate* any noise injection for associative reduce functions. Specifically, instead of moving on to the next reducer if all the key-value pairs for a particular intermediate key cannot be processed by a single reducer (line 5 in Algorithm 1), we permit *splitting* the key-value pairs associated with a key ('boundary key') across two or more reducers, similar to sort-based approaches [12]. This modification requires boundary processing to aggregate partial Reduce outputs for the boundary keys, where each reducer R_i must pass its boundary key and the corresponding aggregated Reduce output to the next reducer R_{i+1}, which computes the final aggregate for the boundary key. This allows us to use $\alpha = 1$ since no fake key-value pairs are required, with execution times comparable to insecure executions while preventing access pattern leakage (§5).

Supporting sort-based and user-defined partitioning. In Algorithm 1, intermediate keys are assigned to reducers in

the order in which they appear in the global histogram, $\hat{h}$. This makes the histogram a straightforward place to support sort-based and arbitrary user-defined partitioning functions. In particular, we order the keys in the histogram based on the user-specified partitioning function, overcoming a key functionality limitation in load balancing schemes (§2.4). If no partitioning function is specified, Weave still requires all weavers to agree on ordering keys in $\hat{h}$ so that they agree on the real/fake key-value allocations across reducers. As such, Weave uses a hash table with the same parameters (*e.g.*, hash function, load-factor, etc.) for $\hat{h}$ at all weavers.

3.5 Weave Overheads

Network bandwidth. An insecure MR framework uses $O(\hat{n})$ network bandwidth during the shuffle phase. In contrast, Weave's network bandwidth usage is $O(\hat{n})$ during the random-shuffle phase, $O(w(w-1)\cdot\beta\cdot\hat{n})$ during the histogram phase, and $O((1+\delta)\cdot\alpha\cdot\hat{n})$ during the balanced-shuffle phase. Note that the histogram phase only broadcasts *sampled key counts*, which are much smaller than actual intermediate key-value pairs. With Weave's default parameters, its network bandwidth overheads are $\sim 3.1\times$ (constant factor) of an insecure baseline: $1\times$ overhead due to the random-shuffle phase and $\sim 2.1\times$ overhead due to the histogram and balanced-shuffle phases (with $\delta = 0.05$ and $\alpha \approx 2$). These overheads are further reduced to negligible for jobs with associative reduce functions. In contrast, prior sort-based and load-balancing schemes incur $\log(\hat{n})$ factor network overhead relative to an insecure baseline, with even higher overheads in practice (§5).

Compute. Compared to an insecure MR framework, Weave incurs additional computations at the weavers during the histogram and balanced-shuffle phases. The computational complexity for the histogram phase is $O(\beta\cdot\hat{n})$ and $O(\hat{n})$ for the balanced-shuffle phase. Again, the added computational complexity for sort-based and load-balancing approaches is $O(\hat{n}\log(\hat{n}))$, with overheads being higher in practice (§5).

EPC Memory Overhead. Weave adds $O(\beta\hat{n})$ of EPC usage during the histogram phase to store $\hat{h}$, $O(\delta\hat{n})$ for padding records after sampling in the histogram phase, and $O(w)$ to hold `kv_real` and `kv_fake` during the balanced-shuffle phase. For non-associative workloads, an additional $O\left(\frac{r}{r+1}\hat{n}\right)$ is incurred to generate and keep fake records on weavers. We note that collectively, this remains a tiny fraction of the total dataset size. In our largest scale experiment with over a billion records, Weave consumes $< 5\%$ of EPC capacity (§5.2) — 0.1% for storing $\hat{h}$, `kv_real`, and `kv_fake`, 1.3% for padding records during the histogram phase, and 2.1% for storing the fake records. While Weave ensures datasets of virtually any size can be analyzed without exceeding EPC capacity in practical settings, Weave can also employ algorithmic techniques to provide memory obliviousness through software-based solutions when EPC capacity is exhausted, discussed in the full version of our paper [26].

3.6 Weave Implementation

Weave is implemented in Apache Spark [2] with an additional 1,500 lines of Scala code. To prevent access pattern leakage in MapReduce jobs, Weave's shuffle phases (random-shuffle, histogram and balanced-shuffle) replace Spark's default shuffle implementation, requiring no modifications to user code.

Weave leverages Gramine LibOS [40–42] for transparent execution atop Intel SGX enclaves. Gramine ensures execution integrity by (i) signing the enclave code, (ii) performing attestation, and (iii) encrypting/decrypting all memory accesses and network communications.

Handling MR jobs with $c > 1$. As noted in §2.4, to meet IND-CDJA security, Weave assumes an upper bound C for the number of intermediate key-value pairs c generated by a `Map` function. To support jobs with $c > 1$, Weave pads every `Map` function's output to C key-value pairs by injecting "filler" key-value pairs. The balanced-shuffle phase processes these filler key-value pairs as fake key-value pairs, i.e., they are distributed evenly among the reducers, where they are dropped without being processed. In our implementation, we assume the user provides this upper-bound C (*e.g.*, for functions like `flatMap`). While most MR jobs have $c \leq 1$, we evaluate the performance impact of $c > 1$ on Weave in §5.4.

Secure EPC design. While EPCs are designed to prevent leakage of memory access patterns, recent works have shown that current *commodity* TEEs such as SGX, SEV, or TrustZone [8–10] are vulnerable to side channel leakages like page-fault monitoring, cache contention, interrupt timing, branch prediction, and speculative execution [43–46].

To protect against such leakages, secure systems built atop publicly available TEEs leverage a *proxy*-based design, where a proxy encapsulates the "leaky" TEE and implements the mitigation of these attacks [33–37, 47–49], retaining memory obliviousness. In Weave, we adopt this proxy-based design, leveraging AEX-Notify [33] to mitigate single-step and interrupt-based attacks and core isolation (similar to Varys [34]) to limit cache timing attacks, all with constant overhead (less than 20%). Moreover, we rely on SGXv2's increased EPC size to minimize page faults for performance.

For more rigorous security guarantees, it is possible to use a formally verified TEE architecture with EPCs that guarantee zero leakage by design *e.g.* [50–52]. We leave the implementation of Weave atop such TEEs to future work.

4 Security Guarantees

This section describes our IND-CDJA security definition (§4.1), contrasts it against prior security models (§4.1), and shows Weave is secure under IND-CDJA (§4.2).

$\text{IND-CDJA}^{\mathcal{A}}_{b,\alpha,C}$:

$n, w, \mathsf{mem} \leftarrow \mathcal{A}_1$
$\mathcal{W} \leftarrow \mathsf{Init}(n, w, \mathsf{mem})$
$D_0, D_1, \mathsf{Map}, \mathsf{Reduce} \leftarrow \mathcal{A}_2(\mathcal{W})$
$o, \tau^N, \tau^M \leftarrow \mathsf{Execute}(D_b, \mathsf{Map}, \mathsf{Reduce}, \mathcal{W})$
$b' \leftarrow \mathcal{A}_3(o, \tau^N, \tau^M)$
Return b'

Fig. 4: $\text{IND-CDJA}^{\mathcal{A}}_{b}$ with adversary $\mathcal{A}$ and random bit b.

4.1 IND-CDJA

The IND-CDJA game, defined in Figure 4, is parameterized by an adversary $\mathcal{A}$, the constant α (which determines `kv_tot`), the maximum Map output size C, and a bit b. The bit b is kept secret from $\mathcal{A}$; the security of a scheme under IND-CDJA is contingent on an adversary not being able to guess b correctly.

In the game, the adversary $\mathcal{A}$ first chooses the input dataset size (n), the number of workers (w), and an array of fixed-size memory cells mem. Based on these parameters, the framework initializes a set $\mathcal{W}$ of w workers, each with the specified mem configuration. $\mathcal{A}$ then picks two challenge datasets D_0 and D_1 of size n, along with a MR job defined by $(\mathsf{Map}, \mathsf{Reduce})$. The framework executes the adversary's chosen MR job on D_b (determined by the random bit b) and generates o, the (partitioned) encrypted output of the job, and a pair of transcripts τ^N and τ^M, where τ^N contains all network communications sent between each worker throughout the job and τ^M contains all reads and writes to memory addresses in mem on each worker; these outputs represent the information that $\mathcal{A}$ can observe from the execution. Finally, given (o, τ^N, τ^M), the adversary $\mathcal{A}$ runs a polynomial-time algorithm to guess b, i.e., which challenge dataset was chosen, and outputs this guess as b'. The adversary "wins" if its guess is correct; conversely, if the guess is no better than a random coin flip, the execution framework is secure.

Prior security definitions. IND-CDJA provides a practical tradeoff between the two main definitions introduced in prior work for secure MapReduce execution. Intuitively, Opaque's obliviousness guarantee is requires the volume of network traffic between all worker pairs to be *exactly* equal. Our IND-based definition aims for more practical guarantees based on the observation that we do not need network traffic to be *identical* for different datasets, only *indistinguishable*. As long as the network traffic distribution is independent of the input data distribution, the adversary cannot infer any useful information. This allows IND-CDJA to cover a larger solution space of schemes, including Weave, without sacrificing security. Although we use a game-based definition in the main text for simplicity, we also provide a composable simulation-based version of IND-CDJA in the full version of our paper [26].

Meanwhile, the notion of 'strong hiding' in the Shuffle & Balance [11] is based on an indistinguishability game, not unlike ours. However, strong hiding allows the MR framework to leak the number of intermediate key-value pairs corresponding to the most popular key, which the adversary can use to infer the key skew of the underlying data and even identify the most popular key [53]. In contrast, IND-CDJA requires this information to be hidden, offering more robust security.

4.2 Proof of Weave Security

We establish the IND-CDJA security of Weave in the theorem below. Weave's security reduces to the pseudorandom security of the PRG shared by its workers, the real-or-random security of its encryption scheme, and the indistinguishability of the memory transcripts generated by its memory access mechanism. As noted in §3.3, Weave uses an oblivious memory pool within the TEE's EPC to secure its data-dependent computations[4]; we assume that the EPC ensures any sequence of accesses to this pool not visible to an adversary (i.e., $\tau^M_{EPC} = \emptyset$).

Theorem 4.1 *For any polynomial-time adversary $\mathcal{A}$ against Weave, there exist poly-time adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that*

$$\mathbf{Adv}^{ind\text{-}cdja}_{Weave}(\mathcal{A}) \leq \mathbf{Adv}^{prg}_{F}(\mathcal{B}) + \mathbf{Adv}^{ror}_{E}(\mathcal{C}) + \mathbf{Adv}^{ind}_{T}(\mathcal{D}),$$

where F, E, and T are Weave's PRG, encryption, and memory access schemes, respectively.

We defer the formal proof of security for Weave with optimizations to the full version of our paper [26], but provide an intuitive proof sketch for Weave without optimizations here. Our proof relies on the network traffic and non-EPC memory accesses of each worker following the same distribution, regardless of the dataset. We outline how each phase achieves this below:

Random-shuffle. Each mapper assigns its intermediate key-value pairs to a weaver uniformly at random, ensuring that the number of key-value pairs sent from each mapper to weaver is uniformly distributed. Also, mappers sequentially access intermediate key-value pairs regardless of their contents, resulting in indistinguishable memory transcripts between datasets.

Histogram. Each weaver broadcasts a histogram padded to the same $\lceil \frac{\hat{n}}{w} \rceil$-sized entries, ensuring that the same communication volume is seen for any dataset with the same number of intermediate key-value pairs. Also, since each weaver W_i keeps the histograms h_i and $\hat{\mathsf{h}}$ in EPC (§3.3), the memory transcript only contains accesses for scanning through the weaver's received key-value pairs for counting. Again, these are sequential and indistinguishable between datasets.

Balanced-shuffle. Consider the distribution of key-value pairs from each weaver across the reducers, where $\texttt{kv_real}[j]$ is the number of real key-value pairs assigned to reducer

[4] Similar to prior work [12], the EPC also ensures obliviousness of Map and Reduce function executions.

Dataset	Properties	Supported Workloads
Enron Email	137M records, 1.7M distinct keys 20B keys and 24B values	HistogramCount, Sort, InvertedIndex
NY Taxi	148M records, 262 distinct keys 4B keys and 4–10B values	HistogramCount, Sort, Median
Pokec Network	31M records, 1.1M distinct keys 4B keys and 4–10B values	PageRank

Table 1: Our evaluated datasets and workloads

R_j. The random-shuffle and balanced-shuffle phase together ensure that the number of real key-value pairs R_j receives from weaver W_i is a binomial random variable, $\hat{n}_{ij}^{real} \sim B(\texttt{kv_real}[j], 1/w)$. Moreover, Algorithm 1 (lines 12–14) ensures that the number of fake key-value pairs R_j receives from W_i is also a binomial random variable, $\hat{n}_{ij}^{fake} \sim B(\texttt{kv_tot} - \texttt{kv_real}[j], 1/w)$. The total number of key-value pairs R_j receives from W_i is then $\hat{n}_{ij}^{real} + \hat{n}_{ij}^{fake} \sim B(\texttt{kv_tot}, 1/w)$, the same distribution across all weaver-reducer pairs. For memory access patterns, each weaver keeps `kv_real`, `kv_fake`, and buffered key-value pairs destined for reducers inside its EPC, hiding any data-dependent accesses. Buffering in the EPC also allows the weaver to send all its key-value pairs at once in a random order, ensuring that the timing and order of transmitted key-value pairs are independent of the data distribution.

Security-correctness tradeoff and α. The game parameter α, which is visible to the adversary, determines the number of key-value pairs that Weave's Execute function sends to each reducer. Theorem 3.2 shows that α must be large enough to ensure correctness. In the unlikely event that maximum key popularity exceeds $\frac{2r}{r+1}$, Weave avoids leaking this information by sacrificing correctness. Specifically, it completes execution with incorrect results (securely notified to the user) by dropping the popular key's real key-value pairs. Weave can subsequently be reconfigured with a larger α. While this reveals that a highly popular key exists in the dataset, our threat model assumes that the adversary already knows the key distribution, i.e., no new information is revealed to it. IND-CDJA's key requirement instead is that *the popularity of individual keys* remains hidden, i.e., the adversary cannot determine *which* key is most popular. Weave thus preserves security as the IND-CDJA adversary cannot distinguish between executions on datasets of the same size and α.

5 Evaluation

We now evaluate Weave against state-of-the-art systems on real-world datasets and workloads.

Compared Systems. We evaluate Weave against three approaches: (i) Insecure baseline that runs tasks within TEEs but provides no network obliviousness guarantees and forms our no-overhead baseline, (ii) Opaque [12], a state-of-the-art oblivious sort-based approach, and (iii) Shuffle & Balance [11], a state-of-the-art load-balancing approach. We implement all approaches in Spark [2] atop Gramine LibOS [41] and make the entire EPC region available for all approaches, with enough memory to ensure memory obliviousness for all their data shuffling requirements. Our ported implementations of prior work closely match the respective system's reported performance results, ensuring a fair comparison.

Experimental setup. Our experiments are conducted on Microsoft Azure cluster with 3 to 20 Standard DC8s v3 instances, each with 8 vCPUs and 32GB of EPC memory. We run all systems inside Gramine LibOS with hardware memory isolation, encryption/decryption, and attestation. We use 10 workers and one controller (11 nodes) as our default configuration, except in our scalability study (§5.2).

Datasets and workloads. We use three real-world datasets: the Enron Email dataset [54], a corpus of corporate emails commonly employed in privacy research [55–58], the NY Taxi dataset, comprising extensive real-world taxi trip records from New York City [59], and Pokec graph dataset [60], comprising the graph for the Pokec social network. All datasets contain privacy-sensitive information with discernible distributions (*e.g.*, email keywords, trip destinations/times, and user profile data like gender and age in the social network) and are known to be vulnerable to access pattern attacks [11].

We employ five analytics workloads across these datasets, although only a subset applies to each dataset (Table 1). The HistogramCount query from the Puma benchmark [61] counts the frequencies of different keys. The InvertedIndex workload maps values in text data to their document occurrences for full-text search efficiency. The Sort job orders records by key. The Median job computes the median of the numerical values for key-grouped records. PageRank is a multistage graph processing workload. Our chosen workloads are representative of practical analytical jobs in privacy-sensitive settings and span diverse complexity ranges and functionality requirements from the MR framework (*e.g.*, non-associative Reduce in InvertedIndex and Median, sorting in Sort, etc.).

5.1 Weave Efficiency

Figure 5 compares the end-to-end execution time for all compared schemes across the various workload and dataset combinations outlined in Table 1 on a 10 node cluster. We note that non-associative jobs (Median and InvertedIndex) and sort-based jobs are not supported by Opaque and Shuffle & Balance, respectively (marked NA in the figure). As a baseline, TEE execution incurs a 1.9 to 2.8× overhead (compare Insecure Baseline with No-TEE Baseline), with the highest cost observed for the associative HistogramCount. The dominant overhead contributors are network and I/O operations, which involve enclave exits and data copies over enclave boundaries. However, these overheads are a necessary minimum for any secure solution.

Weave's execution times are within 1.65–2.83× of the in-

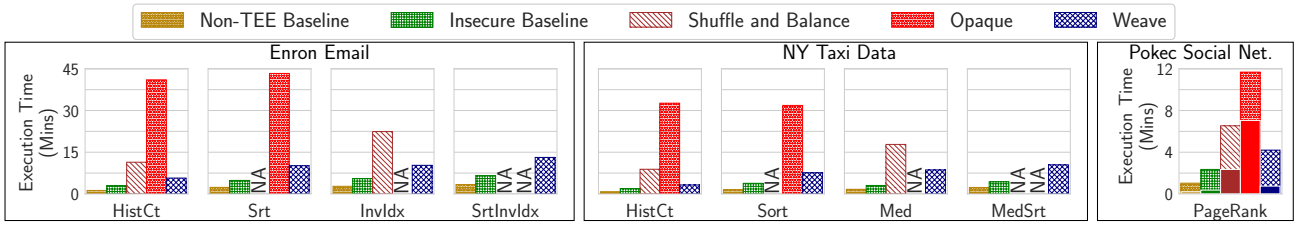


Fig. 5: **Execution time across compared systems & workloads (§5.1).** Jobs that an approach cannot support are marked NA. For PageRank, the solid bars show the time for the first round, while the hatched bars show the time taken for the subsequent 10 rounds.

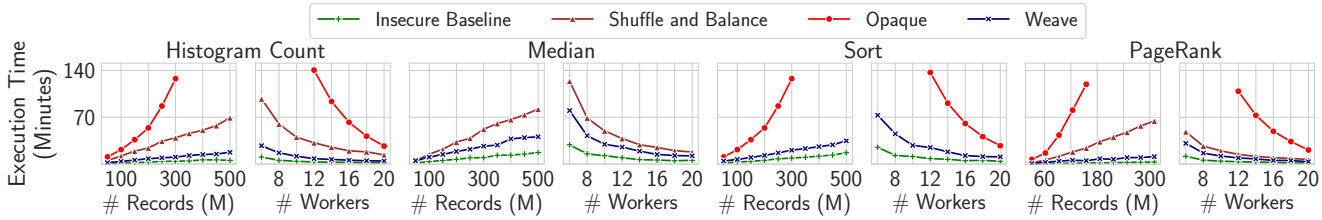


Fig. 6: **Performance scaling with dataset size and worker count (§5.2).** Weave and insecure baseline runtime scales linearly with dataset size, while Opaque and Shuffle & Balance runtime scales super-linearly. The runtime for all four systems reduces linearly with worker count.

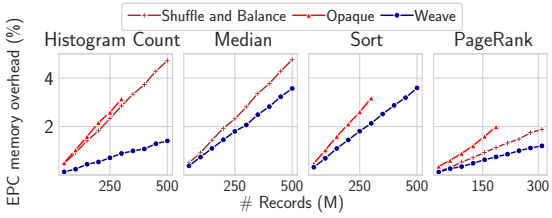


Fig. 7: **EPC memory overhead (§5.2)**w for secure shuffling relative to the insecure baseline scales linearly with data size for all schemes.

secure baseline, while the execution times for Opaque and Shuffle & Balance are 2.8–11.2× and 1.5–5.9× *higher than* Weave. At a high level, most of Weave's performance gains are due to its constant factor bandwidth and compute overheads relative to log-linear overheads seen in compared systems (§3.5). Analyzing the results further yields three interesting observations.

First, the major contributor to execution overheads across all compared systems for almost all workload-dataset combinations is their shuffle approach. Specifically, Weave sees the lowest overhead since Weave's random-shuffle, histogram, and balanced-shuffle phases permit a constant factor overhead relative to the insecure baseline's shuffle phase (1.5–2.7×; not shown in Figure 5). In contrast, Shuffle & Balance's single round of log-linear Melbourne shuffle incurs 3.9–8.3× overhead to insecure shuffles, while Opaque's four-round column sort observes an even higher overhead (7.2–20.2×; not shown in Figure 5). For PageRank, while the first round (solid bars) is dominated by the shuffle required for computing the graph's adjacency matrix, subsequent rounds only require broadcasting without any data shuffle in all systems. Thus, while Weave, Opaque, and Shuffle & Balance are 2.3×, 19.1×, and 9.8× slower than the insecure baseline in the first round, they take the same time for the remaining rounds.

Second, the associative Reduce optimization in Weave permits a 50% reduction in network overheads for associative jobs like HistogramCount. While Opaque employs a similar optimization, the end-to-end execution time remains dominated by its shuffle overheads. We also used mapper-side combiners for the insecure baseline for fairness, although Shuffle & Balance cannot use them since they break obliviousness.

Finally, a balanced distribution of intermediate key-value pairs across reducers in the secure systems, Weave, Opaque and Shuffle & Balance, results in fewer straggling reducers due to a more uniform work distribution across them, *compared even to the insecure baseline*.

5.2 Weave Scalability

We restrict our focus to the NY Taxi and Pokec datasets since they are much larger than the Enron dataset and permit larger-scale performance analysis. We omit Sort-variants for InvertedIndex and Median jobs since prior secure systems do not support them. For PageRank, we restrict our focus to one single round of rank aggregation on the Pokec dataset since subsequent rounds do not involve any data shuffle.

Scaling with dataset size (Figure 6). We fix the worker count to 10 and vary the number of records in the datasets, from 50 to 500 million records for NY Taxi data and from 31 to 310 million records for the Pokec graph data. As expected, the execution time for Weave and the insecure baseline scale linearly with dataset size across all jobs, since Weave incurs at most constant factor overhead relative to the baseline. The

log-linear scaling for Shuffle & Balance and Opaque is also expected due to their shuffle approaches; Opaque's overheads are somewhat higher due to the additional shuffle rounds required for column sort, which gets worse at larger dataset sizes due to memory thrashing during column sort in Spark.

Across various jobs, the computationally light HistogramCount's scaling primarily depends on the shuffle overheads, resulting in a larger relative performance gap between Weave and compared systems. On the other hand, for the more computationally intense Median and Sort jobs, the gap between Weave and compared systems is smaller since the impact of shuffle is overshadowed by the time spent executing the Map and Reduce tasks. At 300 million records, Opaque and Shuffle & Balance observe execution times 9.3× and 2.7× longer than Weave on average, respectively.

Scaling with worker count (Figure 6). We fix the size of the dataset to 200 million records for the NY Taxi dataset and 310 million records for the Pokec social network graph data and vary the cluster size from 6 to 20 nodes. As expected, all four systems scale similarly to the number of workers, i.e., their execution time is reduced proportionally to the number of workers since each of them can split work and network traffic roughly equally among workers. As we saw for scaling with dataset size, the more computationally intensive jobs (Median, Sort) observe a lower performance gap between Weave and other systems in various worker counts, mainly because the impact of shuffle-based compute and network overheads is dwarfed by the time spent in tasks Map and Reduce. At 20 nodes, Opaque and Shuffle & Balance observe execution times 6.7× and 2.3× longer than Weave, respectively.

EPC memory overhead scaling (Figure 7). Our 10-node cluster hosts a total of 320GB of EPC memory. We see that the normalized EPC memory overhead for Weave, Opaque, and Shuffle & Balance — due to additional data structures that must be stored in EPC for each scheme — scales linearly with the number of records in the dataset. For Weave, despite linear scaling, the EPC overhead remains under 1.4% for HistogramCount and 3.6% for Sort on the NY Taxi dataset containing 500 million records, demonstrating that EPC overhead does not pose a scalability bottleneck for Weave.

5.3 Understanding Weave Performance

Weave execution breakdown (Figure 8(a)). The overhead incurred by the random-shuffle phase is purely due to network transfers and is linear in the dataset size. With our sampled histogram optimization, the network and computational overheads of histogram phase are negligible (< 1%). This leaves the balanced-shuffle phase, where the overheads vary by the job's nature and the dataset size. In particular, the balanced-shuffle phase overheads for non-associative jobs (InvertedIndex, Sort, Median) is much higher (up to 40% of the entire overhead) than for associative queries, where our associative reduce optimization all but eliminates compute overheads, and only incurs a network overhead linear in data size.

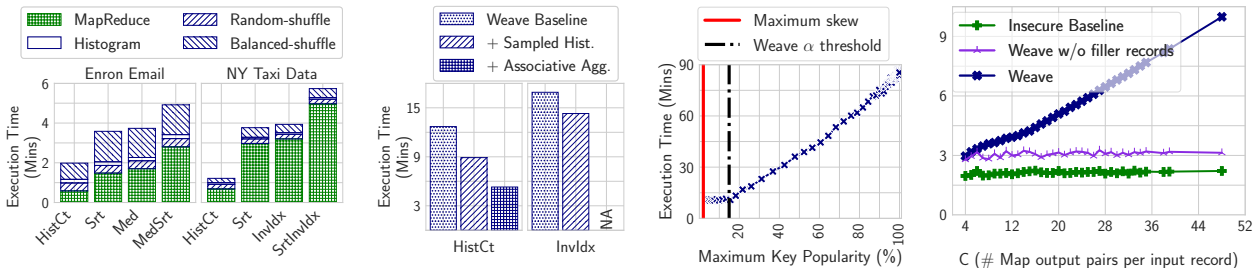
Impact of Weave optimizations (Figure 8(b)). We evaluate the impact of Weave optimizations for the InvertedIndex and HistogramCount jobs on the Enron Email dataset, using the same setup as §5.1. For the Enron Email Dataset, a sampling factor of 0.01 in the sampled histogram optimization reduces the communication time for the histogram phase by 220 seconds. As such, it reduces the execution time for InvertedIndex by 18% and HistogramCount by 30%. The associative reduce optimization only applies to HistogramCount, reducing its execution time further by 33% for a total of 63% reduction.

5.4 Sensitivity Analysis

Impact of α (Figure 8(c)). The choice of α ($\frac{2r}{r+1}$ by default) in Weave imposes a limit of $\frac{\alpha \cdot \hat{n}}{r}$ on the the maximum key popularity (§3.1). While all of our evaluated real-world datasets exhibit a maximum key popularity of < 5% (a small fraction of the threshold set by our default α), we artificially inflate the maximum key popularity in the NY Taxi dataset and evaluate the Median job execution on it to study Weave's sensitivity to larger α values. We use `LocationID` as the intermediate key, inflating the most popular key's popularity while keeping the total dataset size constant.

Figure 8(c) shows that as maximum key popularity (as a % of the entire dataset) increases from 4% to 99% (and consequently, the number of unique keys decreases), Weave's execution time increases by 7.9×. The two vertical lines are added for reference in the figure. The first line shows the maximum degree of skew (3%) in real-world social network datasets for Twitter follower network [62, 63]. The second line, at maximum key popularity of ∼ 14%, shows the point until which Weave's default value α (≈ 1.85) does not need to be increased; beyond this, the minimum viable α required to preserve IND-CDJA security increases linearly up to 13. Interestingly, at the point when Weave's default α needs to be increased, a single reducer ends up processing all intermediate key-value pairs associated with that key — nearly 14% of the entire dataset, and 45% more (real) key-value pairs than any other reducer — an unlikely scenario in real-world settings.

Impact of C (Figure 8(d)). For MR jobs where Map outputs $1 < c < C$ records, Weave (or any other IND-CDJA-secure scheme) must add $C - c$ "filler" intermediate key-value pairs to the Map output (§2.4), increasing its overhead by a factor by $\frac{C}{c_{avg}}$ (§2.4). We evaluate this impact on a Word Count job on the Enron Email dataset, where Map breaks down each sentence into multiple constituent words. We vary the required C by filtering out sentences with word counts over C and scaling up the rest of the dataset to the same size while preserving data distribution. Figure 8(d) shows that as C



(a) Weave execution time breakdown (b) Weave optimizations (c) Sensitivity to α (d) Sensitivity to C

Fig. 8: **Weave performance and sensitivity analysis (§5.3, §5.4).** (a) Random-shuffle phase overhead is linear in dataset size, histogram phase has negligible overheads and balanced-shuffle phase overheads are higher for non-associative jobs. (b) The sampled histogram optimization reduces Weave execution time by 18%–30%; associative reduce optimization reduces it further by 29%. (c) Highly skewed input distributions increase Weave overheads – the red vertical line shows the maximum skew (3%) we observed in real-world social network datasets for Twitter follower network [62, 63]. (d) Increasing C Word Count from 4 to 48 increases Weave runtime by up to 4.1×.

increases from 4 to 48, Weave's performance overhead relative to the insecure baseline increases from 1.32× to 5.34×. Moreover, an insecure variant of Weave (purple line) that does not insert filler records maintains a consistent 1.32× overhead as C increases.

6 Discussion and Future Work

We now discuss areas of future research.

Processing in batches. The complexity of random-shuffle in Weave is $O(\hat{n})$, a key factor in Weave's ability to provide IND-CDJA-security with constant overhead. This, however, requires all of the data generated by the map phases to be randomly shuffled *all at once*. If the data were instead sent in batches, an adversary with prior knowledge of how the input data is divided across them could identify specific data items sent to a weaver based on the batch they were sent in. This is particularly relevant to applications in streaming analytics [64, 65], where MapReduce jobs are executed on *micro-batches* of data. This requires adaptation of random-shuffle phase in Weave to ensure security even when applied to a series of such micro-batches. A possible approach to facilitate this is to adapt the dynamic distribution adaptation schemes and security models explored in prior work on oblivious storage [15, 18] to Weave for obliviousness *per micro-batch*.

Other communication patterns. Our work focuses on shuffle-based all-to-all worker communications in MapReduce analytics. Other analytics frameworks, such as distributed ML training or deep learning, employ different communication patterns, especially in collective operations such as broadcast, all-reduce, gather, all-gather, etc. These communications also suffer from similar access pattern leakages [66, 67] and can benefit from noise-injection-based obliviousness for efficiency and privacy.

Other side channels. While our work focuses on oblivious analytics execution by hiding network and memory access patterns, we consider two additional side channels outside our scope. The first is length-based leakage — similar to prior works [11, 12], we assume the key-value pairs in map-reduce communications are all fixed-sized or can be padded to the same fixed size. However, in real-world datasets, key-value pairs can have sizes that vary by several orders of magnitude, and padding can be prohibitively expensive [68]. Second, we do not consider timing-based attacks, where the duration of the computation can reveal information about the nature of the processing and the data to an adversary. While our security model assumes that the adversary already knows the job being executed, securing such timing-based channels becomes even more crucial when the jobs are secret. We leave the exploration of efficient schemes that can prevent leakage via both side channels to future work.

7 Conclusion

Weave is an efficient and expressive secure analytics platform that scales to large datasets. Weave applies the principles of noise injection to reduce the network and compute overheads for oblivious analytics to a constant factor. Weave reduces the oblivious execution time for analytics jobs on large real-world datasets by 4–10× compared to the prior state-of-the-art.

Acknowledgments

The authors thank the OSDI reviewers and our shepherd for their invaluable feedback. We thank Keen You, Seung-seob Lee, Lin Zhong, and Fan Zhang for their inputs at various stages of the project. This work was supported in part by the NSF's awards 2047220, 2054957, 2112562, 2147946, and a NetApp Faculty Fellowship.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *NSDI*, 2012.

[3] Apache Hadoop. https://hadoop.apache.org/.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[5] Apache Kafka. https://kafka.apache.org/.

[6] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 S&P*, 2015.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI 14*, 2014.

[8] Francis X. McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Hardware and Architectural Support for Security and Privacy*, 2013.

[9] ARM Ltd. Building a secure system using trustzone technology. https://developer.arm.com/documentation/PRD29-GENC-009492/latest/.

[10] AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html.

[11] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *ACM SIGSAC*, 2015.

[12] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

[13] Tom Leighton. Tight bounds on the complexity of parallel sorting. In *ACM STOC*, 1984.

[14] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *ICALP*, 2014.

[15] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, 2020.

[16] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *SIGMOD*, 2015.

[17] Marie-Sarah Lacharite and Kenneth G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018.

[18] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. SHORTSTACK: Distributed, fault-tolerant, oblivious data access. In *OSDI*, 2022.

[19] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. *ACM SIGMOD*, 2023.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SOSP*, 2003.

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE MSST*, 2010.

[22] Intel. Intel sgx. https://www.intel.com/content/www/us/en/developer/articles/technical/overview-of-an-intel-software-guard-extensions-enclave-life-cycle.html. Accessed: 2024-08-30.

[23] Microsoft Resach. Azure security baseline for microsoft azure attestation. https://learn.microsoft.com/en-us/security/benchmark/azure/baselines/microsoft-azure-attestation-security-baseline.

[24] Microsoft Research. Preventing side-channels in the cloud. https://www.microsoft.com/en-us/research/blog/preventing-side-channels-in-the-cloud/.

[25] Raluca Ada Popa. Confidential computing or cryptographic computing? tradeoffs between cryptography and hardware enclaves. *ACM Queue*, 2024.

[26] Mahdi Soleimani, Grace Jia, and Anurag Khandelwal. Weave: Efficient and expressive oblivious analytics at scale. Cryptology ePrint Archive at https://ia.cr/2025/1040, 2025.

[27] Zhilin Zhang, Ke Wang, Weipeng Lin, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. Practical access pattern privacy by combining pir and oblivious shuffle. In *ACM CIKM*, 2019.

[28] Edward G. Coffman, János A. Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: Survey and classification, 2013.

[29] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Cacheshuffle: A family of oblivious shuffles. In *ICALP*, 2018.

[30] Min Xu, Antonis Papadimitriou, Andreas Haeberlen, and Ariel Feldman. Hermetic: Privacy-preserving distributed analytics without (most) side channels. *External Links: Link Cited by*, 2019.

[31] Pengfei Wu, Qi Li, Jianting Ning, Xinyi Huang, and Wei Wu. Differentially oblivious data analysis with intel sgx: Design, optimization, and evaluation. *IEEE TDSC*, 19(6), 2021.

[32] Roberta De Viti, Isaac Sheff, Noemi Glaeser, Baltasar Dinis, Rodrigo Rodrigues, Jonathan Katz, Bobby Bhattacharjee, Anwar Hithnawi, Deepak Garg, and Peter Druschel. Covault: A secure analytics platform. *arXiv preprint arXiv:2208.03784*, 2022.

[33] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. Aex-notify: Thwarting precise single-stepping attacks through interrupt awareness for intel sgx enclaves. In *USENIX Security*, 2023.

[34] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

[35] Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[36] Zhou Hongwei, Ke Zhipeng, Zhang Yuchen, Wu Dangyang, and Yuan Jinhui. Tsgx: defeating sgx side channel attack with support of tpm. In *ACCTCS*, 2021.

[37] Shaizeen Aga and Satish Narayanasamy. Invisipage: oblivious demand paging for secure enclaves. In *ISCA*, 2019.

[38] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 1952.

[39] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[40] Intel. Graphene repository. https://github.com/graphql-python/graphene.git. Accessed: 2024-10-20.

[41] Intel. Gramine repository. https://github.com/gramineproject/gramine.git. Accessed: 2024-10-21.

[42] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library {OS} for unmodified applications on sgx. In *USENIX ATC 17*, 2017.

[43] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598*, 2020.

[44] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution. In *USENIX Security*, 2017.

[45] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *EuroSec*, 2017.

[46] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Sgaxe: How sgx fails in practice, 2020.

[47] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. Mole: Mitigation of side-channel attacks against sgx via dynamic data location escape. In *ACSAC*, 2022.

[48] Qian Ren, Han Liu, Yue Li, and Hong Lei. Cloak: A framework for development of confidential blockchain smart contracts. In *ICDCS*, 2021.

[49] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr. sgx: Automated and adjustable side-channel protection for sgx using data location randomization. In *ACSAC*, 2019.

[50] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[51] Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga. In *ACM CCS*, 2020.

[52] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*, 2020.

[53] Yongzhi Wang, Xiaoyu Zhang, Yao Wu, and Yulong Shen. Enhancing leakage prevention for mapreduce. *IEEE TIFS*, 2022.

[54] CMU. Enron email dataset. https://www.cs.cmu.edu/~./enron/, Accessed: 2023.

[55] Tarannum Zaki, Md Sami Uddin, Md Mahedi Hasan, and Muhammad Nazrul Islam. Security threats for big data: A study on enron e-mail dataset. In *ICRIIS*, 2017.

[56] Fang Liu, Xiaokui Shu, Danfeng Yao, and Ali R Butt. Privacy-preserving scanning of big content for sensitive data exposure with mapreduce. In *ACM CODASPY*, 2015.

[57] Fang Liu, Xiaokui Shu, Danfeng Yao, and Ali R Butt. Privacy-preserving scanning of big content for sensitive data exposure with mapreduce. In *ACM CODASPY*, 2015.

[58] Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. Sedic: privacy-aware data intensive computing on hybrid clouds. In *ACM CCS*, 2011.

[59] New York City Taxi and Limousine Commission. Tlc trip record data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Accessed: 2024-12.

[60] SNAP: Network Datasets: Pokec social network. https://snap.stanford.edu/data/soc-Pokec.html.

[61] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue mapreduce benchmarks suite, 2012.

[62] Wikepedia entry for ml datasets. https://en.wikipedia.org/wiki/List_of_datasets_for_machine-learning_research?utm_source=chatgpt.com.

[63] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *VLDB*, 9(1), 2015.

[64] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM SOSP*, 2013.

[65] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *International Conference on Management of Data*, 2018.

[66] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *ACM CCA*, 2017.

[67] Hanieh Hashemi, Wenjie Xiong, Liu Ke, Kiwan Maeng, Murali Annavaram, G Edward Suh, and Hsien-Hsin S Lee. Private data leakage via exploiting access patterns of sparse features in deep learning-based recommendation systems. In *Workshop on Trustworthy and Socially Responsible Machine Learning, NeurIPS*, 2022.

[68] Grace Jia, Rachit Agarwal, and Anurag Khandelwal. Length leakage in oblivious data access mechanisms. In *USENIX Security*, 2024.



Paralegal: Practical Static Analysis for Privacy Bugs

Justus Adam Carolyn Zech Livia Zhu Sreshtaa Rajesh Nathan Harbison
Mithi Jethwa Will Crichton Shriram Krishnamurthi Malte Schwarzkopf
Brown University

Abstract

Finding privacy bugs in software today usually requires onerous manual audits. Code analysis tools could help, but existing tools aren't sufficiently practical and ergonomic to be used.

Paralegal is a static analysis tool to find privacy bugs in Rust programs. Key to Paralegal's practicality is its distribution of work between the program analyzer, privacy engineers, and application developers. Privacy engineers express a high-level privacy policy over markers, which application developers then apply to source code entities. Paralegal extracts a Program Dependence Graph (PDG) from the program, leveraging Rust's ownership type system to model the behavior of library code. Paralegal augments the PDG with the developers' markers and checks privacy policies against the marked PDG.

In an evaluation on eight real-world applications, Paralegal found real privacy bugs, including two previously unknown ones. Paralegal supports a broader range of policies than information flow control (IFC) and CodeQL, a widely-used code analysis engine. Paralegal is fast enough to deploy interactively, and its markers are easy to maintain as code evolves.

1 Introduction

Applications that handle sensitive user data must comply with privacy policies and legal frameworks like the GDPR [35], access control, and data retention limitations. Even within a single organization, the number of developers modifying a shared codebase on a daily basis makes it difficult to correctly implement and adhere to these requirements [36, 84]. Today, organizations rely on manual audits by privacy experts or external consultants to check if their code respects privacy properties. Such manual audits are laborious, error-prone, and unlikely to happen frequently [69, 72].

Paralegal is a program analysis tool that helps developers find possible privacy problems in their code before deployment. For example, an organization might want to check that all data associated with a user is deleted when they remove their account. With Paralegal, application developers provide meaning to abstractions like "user data" by annotating concrete code elements with *markers*. Privacy engineers meanwhile articulate a high-level policy saying that deletion functions must exist for all types that represent user data, and that the program must call those functions (e.g., "all types marked as user data must flow into deletion functions"). Paralegal models the application by generating its marked Program Dependence Graph (PDG), i.e., a PDG with markers propagated to the appropriate nodes. It then evaluates the policy against the PDG and reports violations to developers.

Paralegal aims to be a practical system for daily use by developers in an integrated development environment (IDE) or a continuous integration (CI) toolchain. While Paralegal is based on a rich literature on bug finding and policy enforcement tools, it contributes two key ideas that improve the practical applicability of such tools on real-world software.

First, Paralegal introduces *markers* as a layer of separation between policy and code that distributes the specification effort between respective experts. Privacy engineers author policies over markers as a vocabulary of semantically meaningful terms, but developers maintain the association of code elements to markers as the code base evolves. This results in policies that are easier to read and robust to code changes. It also allows Paralegal to precisely identify the code relevant to the policy, which it uses to optimize the analysis.

Second, Paralegal leverages the Rust programming language for scalability, precision, and reduced specification effort. Rust's ownership type system controls aliasing and mutation and allows Paralegal to approximate a function's effects by its type. Paralegal leverages such approximations to deal with third party library code when source code is unavailable or when the code is complex and hard to analyze, e.g., as is the case in standard library code. Both these cases traditionally require non-trivial, manual modeling effort. Paralegal further relies on these approximations for scalability. Since markers identify the policy-relevant parts of the codebase, Paralegal cheaply approximates a function's effects via its type signature if no marker is reachable in the function's body or its callees. Rust also encourages static dispatch via its trait system, and aliasing-restricted references via its ownership model. Both make data and control flow analysis, as employed

by Paralegal, more precise.

We have four success criteria for Paralegal: (i) Paralegal should find real privacy bugs, (ii) policies should be expressive, maintainable, and independent from the application code minutiae, (iii) policies should be auditable by non-developers, and (iv) Paralegal should scale to real-world applications.

We evaluate our prototype on eight real-world Rust web applications. Paralegal finds both known and unknown privacy issues in them: Paralegal would have caught five known bugs and found two previously unknown bugs. We compare Paralegal's approach with IFC and CodeQL, a practical code analysis tool supported by GitHub [15]. We find that Paralegal can express a broader range of policies than IFC, that Paralegal finds bugs more reliably than CodeQL, and that Paralegal's markers reduce the complexity of policies compared to CodeQL. We also investigate Paralegal's maintenance effort in evolving applications by applying it to 1,000+ commits spanning 2.5 years of development of Atomic [5], and find that marker changes are rare and no modifications to the policy were needed to keep enforcing it. Finally, we find that Paralegal's optimizations to reduce PDG size allow it to run in seconds, making it suitable for frequent and interactive use.

Paralegal is open-source [33] and our prototype is currently being evaluated for use at a large internet company. This company already has extensive static analysis tooling, but sees value in Paralegal as a complement to it. Specifically, teams at the company are exploring applications of Paralegal to ensure secrecy of cryptographic keys, to enforce encryption-at-rest, and to check that mitigations for speculative execution attacks are executed in hypervisor code.

In summary, this paper makes four key contributions:

1. The Paralegal static analyzer, which checks high-level properties against low-level, evolving code bases.
2. The marker abstraction to decouple policies and code; and techniques to efficiently generate precise marked PDGs from Rust code and model the behavior of library code.
3. A flexible policy framework that compiles policies in a high-level language into queries on marked PDGs.
4. Case studies reporting on our experience of applying Paralegal to eight real-world Rust web applications.

Paralegal has some limitations. As a static analyzer, it can only reason about information known at compile time and must abstract over all possible executions. Paralegal's policy flexibility also means that there is no universally sound approximation during PDG construction. As such, Paralegal's soundness and completeness are policy-dependent (see §4.1.3).

2 Motivation and Background

A practical privacy bug finder must be ergonomic for developers and deal with the realities of real-world codebases, including widespread use of libraries and frequent code change.

Specialized tools achieve practicality by targeting a single domain or type of policy. These systems operate on widely-used programming languages (e.g., Java, Python, Javascript, or SQL) and bake an understanding of the domain into the system—e.g., Android apps in DroidSafe [42], tabular data analytics in PrivGuard [77], or ORM-based MERN (MongoDB, Express.js, React.js, NodeJS) apps in RuleKeeper [38]. Importantly, this domain modeling helps these systems understand the semantics of API functions, libraries, and frameworks without having to analyze their code. However, these tools are limited to checking domain-specific properties and cannot support libraries outside their domain model.

At the other extreme are general **security-typed programming languages**. This category includes languages designed for information flow control [47, 63, 67] and security-typed ORMs [49], as well as proof assistants that encode security policies into dependent types [11]. These languages generally require a purely-functional programming style or extensive annotations on application and libraries, may require manually-authored proofs, and are not widely used in practice.

Prior **code analyzers** let users encode policies as queries over an Abstract Syntax Tree (AST) or a flow-based model of a codebase. Systems that target common programming languages (e.g., C [79], Java [46], PHP [10], Node.js [57], and Ethereum contracts [73]), exist, but all suffer from practicality limitations around library code and policy ergonomics. All of these systems either ignore library code or require developers to provide and maintain manually-written models to convey its behavior. For example, consider CodeQL [9], a "semantic code analysis engine" maintained by GitHub that has backends for languages including C++, Java, and Python. CodeQL's developers maintain extensive, manually-written models of C++ libraries such as `std` and `boost` [16, 17], and users must equally model other libraries they use. Code analyzers also rarely integrate with the codebase being analyzed, but instead encourage policy writers to query syntactic code constructs. For example, CodeQL policies often use regular expressions over identifiers to select source code elements. While helpful for use cases where the goal is to identify syntactic patterns (e.g., retry loops [71]), this design makes CodeQL brittle for enforcing custom semantic properties over changing code.

Paralegal targets Rust, a mainstream programming language whose ownership type system provides Paralegal with the ability to usefully approximate the effects of external library code. Paralegal thus avoids domain-specificity or a need for pervasive flow models to enforce policies over data flowing in and out of library code. Paralegal decouples policies and source code using lightweight markers that developers attach to program elements and maintain with their code.

3 Paralegal Overview

Paralegal catches privacy bugs in Rust programs by extracting a model of the dependency relationships between values at compile time and checking whether this model satisfies a policy written by a privacy engineer. We illustrate Paralegal's workflow using an example based on a real-life bug in Plume,

```rust
#[paralegal::marker(user_data)]
struct Comment { ... }

impl Database {
  #[paralegal::marker(make_delete_query, arguments = [id])]
  fn prepare_delete(&mut self, id: u32, table: &str) {...}
}

impl User {
  #[paralegal::analyze]
  fn delete_user(&self, db: &mut Database) {
    let my_data: UserData = self.get_my_data();
    db.prepare_delete(self.id, "users");
    for post in &my_data.posts {
      db.prepare_delete(post.id, "posts");
    }
    for comment in &my_data.comments {
      db.prepare_delete(comment.id, "comments");
    }
    db.execute();
  }
}
```

Figure 1: *Paralegal alerts developers to missing code (red) to delete a user's comments when deleting their account in Plume [66]. (Code simplified and error handling omitted.)*

```
Somewhere:
1. For each "user data" type marked user_data:
  A. There is a "source" that produces "user data" where:
    a. There is a "deleter" marked deletes where:
      i) "source" goes to "deleter"
```

(a) *The initial Plume user deletion policy.*

```
Somewhere:
1. For each "user data" type marked user_data:
  A. There is a "source" that produces "user data" where:
    a. There is a "deleter" marked make_delete_query where:
      i) "source" goes to "deleter"
      and
      ii) There is a "execute" marked executes where:
        A) "deleter" goes to "execute"
```

(b) *The revised Plume user deletion policy.*

Figure 2: *Example specification for user data deletion written in Paralegal's policy language (§4.3). **Policy clauses**, "variables", and markers are highlighted.*

a federated blogging application [66].

Plume lets users create posts and comments. If a user deletes their account, the application must delete their posts, comments, and the user metadata. Figure 1 shows the user deletion code. `delete_user` on `User` takes a database handle as an argument. It constructs deletion queries for the user themselves and each of their posts and comments, then executes those queries. This function contained a privacy bug: the code in red was missing, so user comments were not deleted [64]. Given a suitable privacy policy, Paralegal catches this bug.

First, a privacy engineer formalizes a policy in Paralegal's policy language (Figure 2a). They express the policy in terms of *markers*. Here, the marker `user_data` describes the concept of personal data, and `deletes` describes the concept of deleting data. Markers allow privacy engineers to formulate the policy at the level of the design of a program, rather than its concrete implementation. The privacy engineer produces the policy in Figure 2a and sends it to a developer.

The developer leverages her application knowledge to apply the policy's markers to relevant program entities. She applies the `user_data` marker to the types `Post`, `Comment`, and `User`. When she tries to apply the `deletes` marker, however, she discovers that there is no correct place to put it. If she applies it to the `prepare_delete` function, the policy would pass if a deletion query is simply *constructed*; nothing ensures that the application actually *executes* the query. She cannot put it on `execute` either: `execute` is a generic function that handles all types of queries, so applying the `deletes` marker here would allow *any* executed query to satisfy the policy.

Since the privacy engineer is unfamiliar with implementation details, they wrote a policy that does not quite fit with the application logic. The developer is faced with a choice: she can either refactor the application to work with the policy as written, or she can work with the privacy engineer to revise the policy. She goes back to the privacy engineer, explains the problem, and both together revise the markers and write a new policy—for all types of user data, there must exist a query to delete them, *and* that query must be executed (Figure 2b). The developer applies the new markers to the application. This give-and-take between privacy engineers and developers is a common Paralegal workflow.

The developer then regularly runs Paralegal to check the policy as she works on the application (e.g., in CI or as an IDE plugin). When faced with the erroneous `delete_user` function, Paralegal reports the error shown in Figure 4. Internally, Paralegal produced the marked PDG shown in Figure 3 and detected that there is no path from a node marked `Comment` to a node marked `make_delete_query` (and from `make_delete_query` to `executes`). The developer fixes the bug by adding the red code in Figure 1, and Paralegal's check passes again.

4 Design

Paralegal is comprised of three components: PDG construction (§4.1) as an abstraction of the program, markers (§4.2) as a semantic vocabulary, and policies (§4.3) that constrain how marked entities in the program can interact.

4.1 Program Dependence Graph

The PDG [37] is a generic representation of a program that can be reliably extracted for applications in any domain. For example, Figure 3 shows a simplified PDG for the Plume application in Figure 1. Paralegal constructs this PDG from Rust's MIR, a control-flow graph (CFG) intermediate repre-

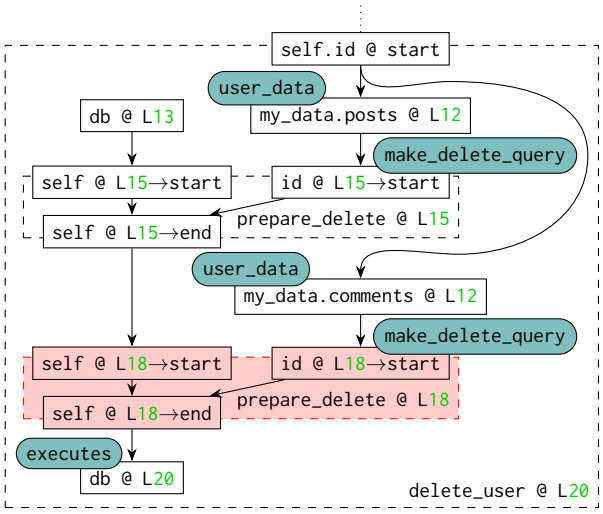


Figure 3: *Partial and simplified PDG for the Figure 1 example. Solid rectangles are PDG nodes, dashed rectangles are function scopes, and teal bubbles are markers. The red subgraph represents the missing comment-deletion code. "Lk" = line k in Figure 1. "start" and "end" refer to the function entry and exit locations, respectively.*

sentation in the Rust compiler.

To check policies on realistic codebases with reasonable accuracy, the PDG must be sensitive to dependencies at a fine granularity. Specifically, the PDG needs three properties:

Flow-sensitivity allows the PDG to distinguish between the values of data at different program locations. For instance, the value of `db` at Line 13 is not the same as the value of `db` on Line 20, so the PDG has separate nodes for each. By contrast, a flow-*insensitive* PDG would represent `db` with a single node, only reflecting the value of `db` at the end of the program, once all of the queries have been prepared and executed. This single-node representation would allow a program that calls `execute` *before* preparing the queries to pass the policy, even though such a program would not actually delete any data.

Context-sensitivity allows the PDG to distinguish between different calling contexts to the same function. Without context sensitivity, the PDG would represent the body of the `prepare_delete` function only once. This would mean that, for instance, its `self` argument is only one PDG node. Each call to `prepare_delete` (Lines 13, 15 and 18) then connects its inputs to that one `self` node in the PDG. It would therefore appear as though `self.id`, `post.id`, and `comment.id` are *all* arguments to the first `prepare_delete` call (Line 13), even though only `self.id` is. As a consequence, the policy could not detect a bug where a developer moves the `execute` call to Line 14.

Field-sensitivity allows the PDG to distinguish between different fields of a structure. For example, a field-sensitive PDG distinguishes `posts` and `comments`, even though they belong to a single variable, `my_data`. A field-*insensitive* PDG would not be able to detect the original bug (shown in red),

```
1  'Deletion Policy' not satisfied.
2  No entrypoints satisfied rule                1.A.a.i
3  Entrypoint `delete_user'
4    Did not satisfy                            (Rule 1.A.a.i)
5      "source" goes to "deleter"
6    for this "source":                         (Rule 1.A)
7      12 | let my_data = self.get_my_data();
8         |     ^^^^^^^
9    for the "user data" type: "Comments"       (Rule 1)
10   Help: There is a "deleter" here:
11     13 | db.prepare_delete(self.id, "users");
12        |                   ^^^^^^^
```

Figure 4: *Paralegal's error for the Plume deletion bug.*

because `my_data` (and therefore `comments`) does flow to `delete`.

Each kind of sensitivity increases the cost of PDG generation for both runtime and memory consumption. Paralegal mitigates these costs with the optimizations discussed in §4.1.2.

4.1.1 Definitions

Formally, the Rust CFG follows a superset of the grammar in Figure 5 (left). In this simplified model, instructions are either assignments, unconditional jumps, conditional jumps, or returns. The left-hand side of an assignment is a *place*, or an expression that refers to a specific region of memory (a variable, a field of a place, or a dereference of a place). The right-hand side can be constants, places, operations on places (including address-of), or function calls. The actual Rust CFG contains more details such as array indexing, but this subset is sufficient to illustrate Paralegal's behavior.

The Paralegal PDG follows the exact grammar in Figure 5 (right). A node in the PDG is a place p at a call string cs, and an edge is an operation of kind ek at a call string. A call string represents the sequence of locations $\ell_0 \dots \ell_k$ that uniquely identify a given instruction in a call tree.

The fundamental goal of the Paralegal PDG is to encode *dependence* within a Rust program. Dependence is a hyperproperty [14] of a program: informally, a variable y depends on a variable x if there exists two executions of a program such that changing x as input changes y as output, with all else held equal. There are two kinds of dependencies: data-dependencies, where the value of x directly affects y (such as $y = x + 1$), and control-dependencies, where the value of x indirectly affects y (such as **`if`** x **`then`** $y = 0$ **`else`** $y = 1$).

A PDG represents dependencies as paths between nodes. Say a PDG contains the edge:

$$p_{\mathsf{src}} \mathbin{@} cs_{\mathsf{src}} \xrightarrow{ek \mathbin{@} cs_{\mathsf{eff}}} p_{\mathsf{dst}} \mathbin{@} cs_{\mathsf{dst}}$$

then it should be the case that p_{dst} at cs_{dst} directly depends on the value of p_{src} at cs_{src} due to the effect ek at cs_{eff}. For example, a PDG for the instruction $y = x + 1$ would contain the edge $x \xrightarrow{\mathsf{data}} y$ (ignoring call strings). If there exists a path from p_{src} to p_{dst} in the PDG, or more formally:

$$p_{\mathsf{src}} \mathbin{@} cs_{\mathsf{src}} \xrightarrow{*} p_{\mathsf{dst}} \mathbin{@} cs_{\mathsf{dst}}$$

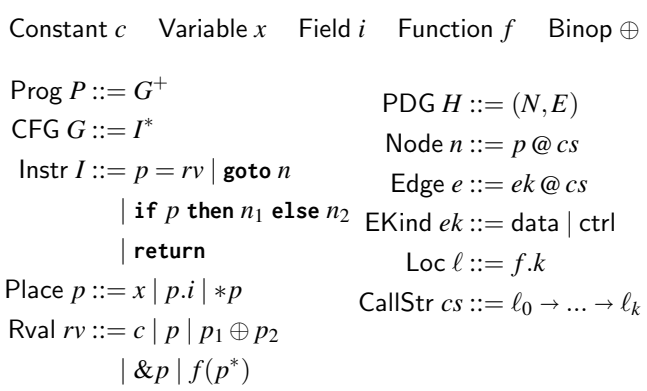


Figure 5: *Grammar for a core subset of the Rust CFG (top and left) and for the Paralegal PDG (right).*

then it should be the case that p_{dst} at cs_{dst} transitively depends on p_{src} at cs_{src}. We use the phrase "it should be" to indicate that this is the ideal case where the PDG faithfully represents dependencies. Paralegal's PDG contains all true dependencies for programs that use no unsafe features (e.g., FFI or pointer arithmetic). §4.1.3 discusses cases outside this guarantee.

4.1.2 Analysis

To construct a PDG, Paralegal statically analyzes each instruction that is reachable from the entrypoints of the analysis. Much of the PDG construction is standard practice—for details, see Ferrante et al. [37]. Our analysis benefits from the choice of Rust as a host language in three key ways.

Monomorphization. Given a function call $f(p_1, \ldots, p_n)$, Paralegal must determine which function f refers to. For example, the expression `x.to_string()` dispatches the trait method `to_string` based on the type of `x`. Paralegal monomorphizes function calls using context-sensitive statically-available type information. For example, if `x` has type `i32`, then Paralegal recursively analyzes the implementation of `<i32 as ToString>::to_string`. Here, Paralegal leverages Rust as the target language for analysis. By design, Rust strongly encourages the use of static over dynamic dispatch—most popular Rust libraries do not use dynamic dispatch at all [74]. Therefore, Paralegal can frequently monormophize function calls, which increases precision around libraries.

Modular approximation. For third-party libraries without source code available, Paralegal cannot know the implementation of f. In these cases, prior systems have either asked users to manually model the behavior of f (e.g., CodeQL maintains a large model of the C++ standard library), or made unsound assumptions about the behavior of f (e.g., Pidgin [46] assumes such functions have no side effects).

Paralegal instead uses the Rust type system to approximate the behavior of f soundly and precisely. At its core , the approximation conservatively assumes that all arguments to a function will influence all outputs of the function. However, Paralegal builds on the technique used in the Flowistry information flow analyzer for Rust [28], which provides two key pieces of information that refine this approximation:

1. *Mutability:* In Rust, a program is not allowed to mutate data accessible from an immutable reference. Paralegal can therefore assume that function calls do not mutate values behind immutable references, which limits the set of plausible outputs and therefore the data flows introduced by the approximation. For example, Rust's `HashMap<K, V>` has a method (slightly simplified) for removing a value:

   ```
   fn remove(&mut self, key: &K)
   ```

   Paralegal can assume that `HashMap::remove` only mutates `self` and not `key`.
2. *Aliases:* In most languages it must be assumed that, in addition to the described data flows, a function may introduce arbitrary aliasing relationships on pointers it has access to. In practice, such an assumption causes too many false positives to make for a useable analysis. In Rust however, all references (the most common pointer-like type) are annotated with a *lifetime* that indicates a precise, limited set of possibly aliased objects. For example, `HashMap<K, V>` has a method for getting a value by key:

   ```
   fn get<'a, 'b>(&'a self, key: &'b K) -> Option<&'a V>
   ```

   This method returns a reference to a value of type `&V`. Because the reference has lifetime `'a` and not `'b`, Paralegal can assume that `&V` points to `self` and not `key`.

This *modular approximation* of a function's behavior is highly accurate compared to precise analysis with access to the function's source code [28]. Therefore, this technique helps Paralegal retain precision while reducing developer burden.

Similarly, resolving function calls with dynamic dispatch is challenging. Paralegal leverages the same modular approximation to approximate the effects of a function behind dynamic dispatch; with additional engineering, Paralegal could analyze all possible function bodies to improve soundness [4].

Function cloning. Paralegal achieves context-sensitivity via function cloning [78], where each call-site duplicates the sub-graph of the called function. This technique is maximally precise, but it risks exponential growth in the size of the call graph. We found in practice that PDGs for realistic Rust applications are nonetheless small enough such that Paralegal runs reasonably fast, even for codebases with 198k LOC (§7.4). Two key factors enable this scalability. First, Paralegal uses markers to reduce PDG size, which we will describe in §4.2. Second, Paralegal uses a context-insensitive alias analysis based on Rust's lifetimes, which allow Paralegal to reuse the PDG for a function across all call sites to that function.

4.1.3 Limitations

Paralegal's PDG construction is similar to most static analyses in that it may include false dependencies by abstracting away relevant details of the code. For example, Paralegal will assume a branch could always reach both targets. This causes

a false dependency in cases like `z = if e { x } else { y }` where `e` is an expression that always evaluates to false. The impact of such inaccuracies on soundness or completeness depends on the policy being checked. For example, including a false dependency can cause a false-positive for policies like "secure sources *cannot flow* to insecure sinks," while it may cause a false-negative for policies like "user data *must flow* to a deletion function."

Paralegal errs on the side of including false dependencies rather than omitting true dependencies. Specifically, Paralegal's guarantee is that in Rust code without unsafe blocks, all true dependencies will be reflected in the PDG.

In general, developers should treat Paralegal output similar to that of a linter or bug finder. That is, a developer should not assume, "if Paralegal says my app is okay, then it is 100% secure or bug-free." Rather, the developer should assume, "if Paralegal says my app contains a policy violation, then I should investigate it." Some sources of inaccuracy are incidental and can be addressed with further engineering (see §5). Others are fundamental and discussed next.

Unsafe code. Rust uses `unsafe` blocks to permit operations that cannot be verified safe by the compiler, including FFI (e.g., calling C libraries) and use of raw pointers (as opposed to compiler-checked references). Paralegal may omit true dependencies induced by unsafe code, such as aliases induced by pointer arithmetic, because Paralegal reasons about aliasing via Rust's lifetimes.

Paralegal mitigates this limitation by using its type-based approximation. A common pattern in Rust FFI is to carefully encapsulate unsafe code within an API presenting a safe interface. Paralegal can analyze such an API just at the interface level without observing the unsafe internals.

Interior mutability. Rust provides "interior mutability" primitives like `RefCell<T>` which permit mutating data behind an immutable reference—that is, one can turn an `&RefCell<T>` into an `&mut T`. This special case violates the assumption of the type-based approximation that a function cannot mutate immutable references. Therefore, Paralegal may omit true dependencies when approximating calls to functions using interior mutability. This limitation also extends to shared-memory constructs, such as locks and mutexes.

This limitation is mitigated by two factors. First, interior mutability is rare in idiomatic Rust. For example, the largest application in our evaluation (HyperSwitch, about 198k LOC) contains no interior mutability in its application logic. Second, interior mutability impacts Paralegal's type-based approximation, not the dependency analysis. For example, a directly observed mutation `*cell.borrow_mut() = 1` *is* registered correctly as mutating `cell`, but if it takes place in `fn foo(cell: &RefCell<T>)` and `foo` get approximated, it is not.

External effects. Paralegal may omit true dependencies induced by effects on external systems like a file system, OS, or database. For example, if `f` is a `File` called `foo.txt`, then Paralegal understands that `f.write(bytes)` affects `f` because `write` requires a mutable reference to `f`. But Paralegal does not understand that `f2 = File::open("foo.txt")` is effectively an alias on `f`, and that `f2.read()` should depend on `f.write(bytes)`.

4.2 Markers

To articulate privacy policies in terms of a PDG, privacy engineers need to refer to nodes of interest. Most prior systems keep policy-related information entirely outside the codebase, and consequently require policies to embed direct references to functions or types. Such direct references make policies more complex and brittle, as §7.2.2 will show.

Paralegal instead uses *markers* to abstract categories of related nodes in the PDG. A marker is an abstract label, such as "sensitive data" or "public sink." Application developers attach markers to code elements, either via source-level annotations (e.g., `#[marker(user_data)]`) or an external configuration file. Developers can mark functions, function arguments, return values, and type definitions. The basic concept of a marker is relatively straightforward, but the subtleties lie in two areas: how Paralegal propagates markers to the PDG, and how Paralegal uses markers to optimize PDG generation.

Marker propagation. After generating the PDG, Paralegal assigns markers to the concrete nodes in the PDG that represent the abstract marked code elements. A parameter p of f, marked with m is a straightforward case. Paralegal assigns m to all PDG nodes that represent the *actual* parameter p at call sites of f. Paralegal handles markers on return values of functions analogously.

The type case is more complex. The simplest situation is as follows: say a marker m is attached to a type τ, and say a node $n = p \,@\, cs$ has $p : \tau$. Then Paralegal propagates m to n.

However, this algorithm may not always capture a privacy engineer's intent. For example, say a `Password` type is marked `sensitive`, and a `User` struct contains a field of type `Password`. Then say the code has a variable `u : User` and calls `f(&mut u)` on some black-box function f. The modular approximation will create one node for u at this call site, but not a node for every field in u.[1] The simple propagation strategy therefore neglects to attach m to n_u. However, the privacy engineer likely expects that because `Password` *could* have been mutated by f, the node n_u should be treated *as if* it were of type `Password`. To capture this expectation, Paralegal's actual algorithm propagates the marker. Say a node has $p : \tau'$. If marked type τ appears anywhere within τ' (e.g., the `User` struct, or $\tau' =$ `Vec<`τ`>`), Paralegal propagates m to p.

Adaptive Approximation. In Paralegal, policies can only talk about application entities that are marked. Paralegal may therefore soundly assume that only code interacting with marked entities influences policy enforcement outcomes. Paralegal leverages this assumption when analyzing an instruction that calls a function f by checking if there are markers reach-

[1] Such an approach would work, but increases the size of the PDG and leads to no additional precision, since function signatures are not precise enough for a field-sensitive approximation.

able from the body of f, and if not, approximates f's effects via type signature instead of generating its subgraph. Paralegal determines reachable markers by visiting all code reachable from f with a cheap call graph traversal. Paralegal fully monomorphizes f and all other functions during the traversal to prevent ambiguity as to what code is called for trait methods. Paralegal caches and reuses the results of marker-reachability traversals for each function. This optimization substantially improves PDG generation speed (§7.5).

4.3 Policies

Paralegal policies are assertions about either necessary or impermissible paths in the marked PDG. For example, a high-level privacy policy such as "a user must be able to delete all of their user data" can be encoded as the assertion "for all types marked `userdata`, there must exist a path from a node with that type to a node marked as a deletion function."

To express such assertions, Paralegal provides a policy language with a controlled natural language syntax that mimics the structure of legal documents. Paralegal compiles policies into Rust programs that use a low-level API to query the marked PDG. The primitives of the DSL are markers, variables, and PDG relations. Primitive relations include:

- `a "value" marked sensitive`
  binds PDG nodes marked `sensitive` to the name "value".
- `"value" goes to "sink"`
  Checks if a *data flow* path from `"value"` to `"sink"` exists in the PDG.
- `"value" affects whether "operation" happens`
  Checks if `"operation"` has a *control flow* dependency on `"value"`.
- `"value" goes to "sink" only via "disclosure"`
  Holds if every data flow path from `"value"` to `"sink"` contains at least one node in the set `"disclosure"`. This primitive allows policies to describe instances of declassification.

Privacy engineers than compose these primitives in first-order logical formulas. For example, a formula like this:

$$\forall x \in X.\ \exists y \in Y.\ P(y) \implies (P(x) \vee S(x,y))$$

is expressed in the policy language as the program on the left, which Paralegal compiles into the Rust code on the right:

```
1. For each "x" in X:
  A. There is a "y" in Y where:
    a. If P("y") then:
      i. P("x")
      or
      ii. S("x", "y")
```

```
X.iter().all(|x|
  Y.iter().any(|y|
    !P(y) || P(x)
          || S(x, y)))
```

The policy language requires policy writers to delineate the scope of each line in their policy with clauses (e.g., 1, A.). This structure of nested clauses, inspired by legal writing, seeks to achieve readable and unambiguous properties. The policy language is decidable because all primitive relations are decidable (the PDG is finite, so reachability is decidable), all quantifiers are decidable (they range over finite subsets of nodes), and recursive policies are inexpressible. Appendix A.3 has the full grammar for the policy language.

Paralegal allows policy writers to author policies as graph queries using the low-level Rust API. This primarily helps policy writers customize error messages, for example by emitting traces and paths through the PDG.

4.4 Error Messages

Paralegal's error reporting helps developers diagnose a failing policy. Error messages print the violated rules and the source code location that instantiated a given quantified variable (as in Figure 4). Paralegal reports them with a diagnostics framework inspired by the Rust compilers's error messages, relating graph nodes to snippets of the source code. Policies written with the low-level API can use this diagnostics framework directly to create customized error messages.

5 Implementation

Our Paralegal prototype consists of 15.1k lines of Rust and is implemented as a Rust compiler plugin.

Multi-crate support. Paralegal extends analysis across multiple crates by persisting the Rust MIR, the "outlives" relationships of lifetimes, output of the Rust type checker, and any Paralegal marker annotations. In one of our largest case studies (the social media application Lemmy [55]) this metadata for all crates combined is 411MB. (For context: `rustc` produces 258MB of metadata for the same application.) During PDG construction, Paralegal lazily loads the MIR for reachable functions and generates their PDG if markers are reachable. Paralegal's PDGs can span all crates for which `cargo` initiates compilation in the process of building the target application. Developers can limit this set for performance. This approach preserves source location information and error messages may reference locations in any loaded crate. A PDG cannot extend to external shared or precompiled libraries; in such cases, Paralegal uses the modular approximation.

Await. Paralegal deliberately discards all *control flow* introduced by the state machine created by `await`. This control flow is needed for Rust's async runtime, but it causes confusing false positives because privacy engineers do not expect these dependencies due to their transparent nature. The consequence of removing them is more predictable policy behavior at the cost of being unable to check for certain malicious async patterns, such as futures that hang indefinitely.

Marker limitations. Paralegal's marker annotations currently only support functions, function arguments, function return values, and type definitions. While these represent the common boundaries for semantic meaning of source code elements, policies would be more ergonomic if Paralegal allowed markers to be attached to fields of a type or global constants. Expanding the set of markable elements is feasible with additional engineering.

6 Case Studies

We now discuss our experience applying Paralegal to eight real-world Rust applications. We tried to pick popular, pro-

Application	Type	LoC	Policies	Unique Markers	Marked Locations	Entry pts.
Atomic [8] (v0.34.2)	Graph DB	9.6k	Access Control	4	4	1
Contile [27] (v1.11.0)	Advertising	4.9k	Purpose Limitation	3	5	1
Freedit [41] (v0.6.0-rc.3)	Social	6.6k	Data Retention/Expiration	5	5	4
Hyperswitch [44] (v0.2.0)	Payments	198.9k	Credential Security, Limited Collection	6	7	3
mCaptcha [59] (v0.1.0)	Authentication	10.6k	Data Deletion, Limited Collection	5	5	2
Lemmy [55] (v0.16.6)	Social	31.4k	Access Control	8	145	72
Plume [66] (v0.7.2)	Blogging	21.4k	Data Deletion	7	7	1
WebSubmit [68] (v1.0)	Homework	1.6k	Data Deletion, Access Control	11	18	3

Figure 6: *Case study applications with code size, policies, and Paralegal marker statistics. "Marked Locations" indicates the number of source code entities (arguments, returns, etc.) we marked; "Entry pts." is the number of analysis entry points.*

duction-level Rust applications that cover a range of domains and coding styles (Figure 6). The source code for these applications exercises language features and data-structures typically found in realistic software, including loops, traits, type parameterization, closures, higher-order functions, error handling, vectors, hash maps, strings, and async. We marked between four and 145 program locations, roughly proportionally to the amount of application code covered by policies.

Policies cover privacy properties from classic access control and security to data deletion and expiration, as well as purpose limitation (rules about what purposes data can be used for). We formalize 11 policies (1–2 per application), each with several clauses and nesting up to three quantifiers per clause (median: 3). The source code for all case studies, as analyzed in this paper, is available in our artifact [3].

6.1 Policies

We initially developed policies by examining source code, then directly writing policies about the expected flows in the PDG. But these policies used more markers and were more closely tied to source code than necessary. Hence, we switched to defining the policy first, using application functionality and documentation (as a privacy engineer would), then applying those policies to source code.

We found this approach made it easier to define policies, and that these new policies were clearer, more concise, and more portable to other applications. Privacy-related properties of applications are often obvious from the UI, functionality (e.g., account deletion), or documentation. By contrast, navigating a large, unfamiliar codebase to search for privacy-relevant sections is much harder. This experience inspired us to design the policy DSL. In two of eight cases we studied (see §6.3) our DSL policies, written without knowledge of source code, matched application semantics without revision.

We found that Paralegal is expressive enough to represent all the policies we wanted to check. Two applications (mCaptcha and Plume) use identical data deletion policies, except for their application-specific marker names. For policies that are fundamentally dynamic, we were able to define static approximations. For instance, Freedit, a social media platform, stores a user's viewing history, but deletes the data after three days. Since the current time is only available at runtime, Paralegal cannot directly verify that Freedit obeys the three day expiration limit. However, it can approximate this policy by checking that `viewing_history` flows to an `expiration_check` and that `expiration_check` has control flow influence on a `deletes`.

6.2 Markers

We found it easiest to apply markers to applications with modularized code that has clearly defined semantics. Applications with specialized delete or authorization check functions were simpler to apply markers to than applications that inline such logic inside large functions. We also found that many of our markers (e.g., `user_data`) could be applied to types alone. Idiomatic Rust programs often define fine-grained, custom types that are clearly named for the type of data they represent.

We mainly made two types of source code change. The first is because our prototype cannot apply markers to constants or to fields of a type, so we defined no-op functions `mark_{marker_name}(&data)`, whose sole purpose is to apply the appropriate marker to `data`. Second, if applications inlined privacy-critical functionality inside a larger function, there was no way of applying a marker to just the relevant lines. While this is not fundamental—a more complex policy could handle it—we extracted the logic into a helper function and marked that function. Via this process, Paralegal encouraged us to cleanly demarcate privacy-relevant code.

Appendix A.2 describes four other minor changes we made.

6.3 Selected Experiences

Two cases in our experience of developing policies stand out.

Atomic is a graph database that lets users create, edit, and share graph-structured data [5, 8]. Each time a user modifies a database resource, Atomic stores a signed *commit* record. Before creating a commit, the application must verify that the user has permission to modify that resource. Crucially, this authorization must happen *before* updating the database resource [7]. Our Paralegal policy asserts that a resource that will be modified first flows to an authorization check, which has control flow influence on the modification.

When we first ran this policy on the application, it failed,

Application	Bug Description	Reference	Paralegal Policy
Plume [66]	Comments and Media not deleted when a user is deleted.	commit 19f1842, issue 806	If a user flows into a delete function, all types marked as user data flow into a delete function.
Atomic [8]	Users can grant themselves write access to data without prior access.	commit 46a503a	Write permissions must be checked before a resource is updated.
Lemmy [55]	Banned or deleted users can log into a server.	commit b78826c	(1) A user deletion check and a ban check must influence every database access, except those reading the active user. (2) Community deletion check and user ban check must influence every database write to a community.
	Deleted users can perform actions in a server.	commit 2966203	
	Users can write to a deleted community.	commit 2402515, issue 2372	
	Banned users can act in communities.	issue 2372	

Figure 7: *Paralegal found seven bugs, including two previously unknown ones, in three applications: Plume [66], Atomic [8], and Lemmy [55]. The rightmost column summarizes, in prose, the Paralegal policy we used to find each bug.*

even after the developers fixed the bug. Upon investigation, we realized that we had missed an edge case: if a resource is new, the application *first* modifies the resource to set default permissions, *then* executes the permission check. We missed this logic initially because it was unclear in the application's documentation. In this situation, we had two options: *(i)* change the policy to exempt modifications that don't depend on user-provided data; or *(ii)* extract this benign modification into a helper function with an exception marker, and exempt this marker in the policy. The first choice burdens the policy with matching a specific code pattern; the second (which we chose) exempts only a specific code instance and clearly delineates the exception in the code itself.

mCaptcha is a proof-of-work (PoW) CAPTCHA service focused on privacy [59]. Website owners register sites with mCaptcha and embed mCaptcha API calls into their sites. mCaptcha's PoW algorithm features a tunable "difficulty", designed to balance security and latency.

In addition to adding a data deletion policy, which we reused from Plume (Figure 2a), we observed that mCaptcha collects optional consent to "gather performance statistics [...] and make them available to other mCaptcha installations" for difficulty tuning [45]. We wrote a policy to enforce that mCaptcha checks whether a user has opted in before storing their data, but the code failed this policy.

On discussing the issue with the mCaptcha developers, it turned out that we had misinterpreted their privacy goals—statistics are always collected, even if website owners don't opt into sharing them. The developers, however, indicated that they would be open to reconsidering this choice [58]. In addition, the discussion helped the mCaptcha developers find a (related) data integrity bug: they deleted all statistics when a user revoked their opt-in consent, rather than just the consent. This illustrates how the discipline enforced by Paralegal helps developers reflect on their code and find issues.

7 Evaluation

Our evaluation of Paralegal seeks to answer five questions:

1. Does Paralegal find bugs that result in privacy violations in real applications? (§7.1)
2. How does Paralegal compare to IFC and CodeQL in terms of expressiveness, practicality, and ergonomics? (§7.2)
3. Does Paralegal's decoupling of code and policies improve ergonomics for evolving applications? (§7.3)
4. Is Paralegal fast enough for interactive use, and how does its runtime scale with the amount of code analyzed? (§7.4)
5. How does Paralegal's adaptive approximation optimization contribute to its effectiveness and speed? (§7.5)

Setup. All experiments run on a server with an Intel Xeon E3-1230v5 CPU (3.4 GHz) and 64 GiB RAM, on Ubuntu 20.04 using Rust nightly-2023-08-25.

7.1 Finding Privacy Bugs with Paralegal

We applied Paralegal to eight applications (Figure 6) to investigate its ability to discover bugs. A good result for Paralegal would show that it finds previously reported bugs as well as new bugs, without generating many false positives.

Figure 7 summarizes the bugs Paralegal found: Paralegal found two previously unknown privacy bugs that were confirmed by the developers, as well as five previously known privacy bugs that the developers had already fixed.

Atomic and Plume. Paralegal found three known bugs in Plume and Atomic [6, 64, 65]. In Atomic, the policy passes Paralegal after the fix [6]. In Plume, however, the policy still fails after the developers' fix [64], since even though the application now deletes comments correctly, it still fails to delete users' uploaded media (a separate, known bug [65]).

Lemmy. We ran Paralegal on 72 HTTP endpoints in Lemmy. Paralegal found two bugs previously fixed by the Lemmy developers and two new additional bugs.

Known Bugs. A user may not access a Lemmy instance if their account has been banned or deleted. However, Lemmy's helper for authorizing already logged-in users omitted a check for whether their account is deleted [50]. Consequently, Paralegal flagged the instance authorization policy in all endpoints. After the fix [30], 71 endpoints passed the policy, but Paralegal still reported a failure in the login endpoint. Since the the login endpoint doesn't use the authorization helper for logged-

Application	Policy	IFC	CodeQL	Paral.
Atomic	Authorization	✓ 🔧	✗ †∗T¶	✓
Plume	Data Deletion	✗	✓	✓
Hyperswitch	Credential Security	✓	—	✓ 🔧
Hyperswitch	Limited Collection	✓ 🔧	—	✓
Websubmit	Data Deletion	✗	✗ ‡A	✓
Websubmit	Access Control	✗	✗ ∗	✓
mCaptcha	Data Deletion	✗	✓ T¶	✓
mCaptcha	Limited Collection	✓ 🔧	✓ T¶	✓
Freedit	Data Retention	✗	✗ ‡	✓
Lemmy	Access Control	✓ 🔧	(✓) †¶	✓
Contile	Purpose Limitation	✓	—	✓

Figure 8: *Paralegal expresses and enforces properties that baseline approaches (classic IFC and CodeQL) struggle with. ✓ indicates success, (✓) success on some versions, ✗ failure; 🔧 denotes required code changes beyond annotations, and we indicate CodeQL results affected by †control flow analysis, ‡hidden source code, ∗taint propagation to/from structures, Aalias analysis, Tunconstrained templates, and ¶async code. "—" means the application was too large to translate to C++.*

in users, it still did not check for account deletion [53]. The login endpoint was also missing a check if the user had been banned. After a second fix by the Lemmy developers [31], all 72 endpoints passed the policy.

New Bugs. Lemmy prohibits users to write in deleted communities: if a community was removed for problematic content, for example, users must not be able to make new posts. The Lemmy developers already found missing community deletion checks in five endpoints [32, 51], but Paralegal found 16 further endpoints lacking these checks.

In addition, a banned user should not be able to write to a community. Paralegal reported that some Lemmy controllers are missing these community ban checks. This allows bypassing access control: for example, a banned community moderator can immediately unban themselves. The Lemmy developers confirmed both bugs [52, 54].

7.2 Comparison with Related Work

Next, we evaluate how Paralegal's expressiveness and ergonomics compare to prior approaches, using the eleven policies from our case studies. We consider two baselines: *(i)* classic IFC based on a lattice of security labels; and *(ii)* CodeQL, a recent code analysis engine deployed at GitHub [15]. We tried to express the policies of our case study applications for both baselines. In IFC, this boils down to imposing a label hierarchy and determining declassification points. For CodeQL, we implemented appropriate queries.

7.2.1 Comparison with IFC

In classic IFC, labels applied to data form a lattice, and the system ensures that data with low-security labels is free of influence from values with high-security labels (the "non-interference" property). Some security and privacy concerns fit into this model, but its expressivity is limited compared to generic tools (CodeQL and Paralegal). As such, we expect IFC to be able to express some, but not all policies.

Figure 8 shows that IFC can enforce six of the eleven policies. Two properties, in Contile and Hyperswitch, directly fit IFC's notion of restricting flows of high-security values into low-security sinks. Policies in Lemmy, Atomic, Hyperswitch and mCaptcha require access control or consent checks prior to operations on data, which IFC can approximate via selective declassification. However, this strategy requires code changes beyond the usual addition of annotations (🔧); in this case, turning the check into a data flow operation, as IFC cannot declassify (i.e., remove a label) via control flow. WebSubmit's Access Control policy is a complex, data-dependent property that requires a list to contain only "blessed" receiver email addresses. Since IFC never prohibits upgrading a security label (e.g., from the low security "external" to high security "blessed"), IFC alone is insufficient to enforce this policy. Disciplined use of data types, combined with IFC, can enforce this property, but would require a substantial code rewrite. (Freedit's Data Retention policy contains a similar pattern.) Finally, Data Deletion and Retention policies rely on a "must reach" pattern that requires a value to reach a sink (i.e., deletion). IFC can enforce safety but not liveness properties, i.e., it can only check the absence of prohibited flows, but not mandate the existence of data flows. Thus, IFC fails to express and enforce these policies.

7.2.2 Comparison with CodeQL

CodeQL extracts a program's AST and derived information into a database, and developers write queries against it in a Datalog-like language. Since CodeQL lacks support for Rust, we translated the relevant parts of each application into C++, the most similar language to Rust with CodeQL support. Two applications (Hyperswitch and Contile) have policies that touch large amounts of code (1.3–1.5k LoC, plus library functions), so we omit them. For Lemmy, we translate one representative endpoint, but omit 71 structurally similar ones. Our ports seek to faithfully reproduce the control and data flow of the Rust code, but replace Rust's `Result` with C++ exceptions. We implement the queries using CodeQL's libraries for data flow [18] and control flow analysis [19].

CodeQL's query language can express all eight relevant policies, but its analysis engine fails to enforce some of them; when this happened, we debugged the issue sufficiently to pin it down to one or more limitations in CodeQL or its libraries, and stopped investigating further (i.e., our analysis may have missed further problems masked by the initial failure).

Policy Effectiveness. Figure 8 shows the outcome of each CodeQL policy. Plume's Data Deletion policy works; as do mCaptcha's policies (with caveats, see below) and Lemmy's Access Control policy in some versions of the code. Other properties fail due to combinations of limitations in CodeQL,

identified by symbols:

(†) Control flow analysis in CodeQL is not inter-procedural. This causes a false positive with Lemmy's access control policy, which uses a helper function for access control checks. CodeQL's control flow analysis also (intentionally [21]) ignores certain complex control flow patterns [22], which causes a false positive in Atomic.

(‡) CodeQL assumes no data flows exist through library calls whose source code is unavailable. This affects Websubmit's Data Deletion policy, which hinges on detecting data flow into a value-type constructor provided by a database library; similarly in Freedit's Data Retention, CodeQL misses data flow through a database method. While CodeQL allows developers to write manual models for library functions, this is onerous and error-prone.

(*) Taint propagation to fields within a structure and from fields to the structure itself requires manual modeling [20]. This complicates policies over markers on a structure that expect to detect flows of that structure's fields into a sink; or policies over markers on elements of a container that expect to detect flows of the container itself. The CodeQL developers manually model taint propagation for collection elements for some standard library data structures but do not (yet) model `std::unordered_map`, which Atomic and Websubmit's Access Control use.

(A) CodeQL currently lacks an alias analysis for C++, due to performance issues [24]. As a workaround, the developers manually model methods, e.g., for `std::vector::push_back`, but CodeQL lacks a default model for initializer lists, which show up in Websubmit's code checked by the Data Deletion property, and require reasoning about aliasing.

(T) Our C++ ports replace trait-constrained Rust types with C++ templates. CodeQL cannot analyze code with uninstantiated template parameters [23], even when constrained by concepts. By contrast, Paralegal can analyze uninstantiated, parameterized code by exploiting trait constraints with the modular approximation.

(¶) C++ has no language-level async support, but provides an `async` library function. When we use it in place of Rust's async, CodeQL misses data and control flow dependencies through this function [25], causing false positives and false negatives in Atomic, mCaptcha, and Lemmy. As a workaround, we use synchronous C++ code.

The last two limitations in part result from impedance mismatches between C++ and Rust, so we report them in Figure 8, but still consider the affected policies to succeed.

By contrast, Paralegal has an inter-procedural control flow analysis, models the effects of unknown library code via modular approximation, collapses taints in and out of structures without developer effort, and relies on Flowistry's modular alias analysis for Rust. It also handles uninstantiated type variables constrained on traits by approximating the effect of trait methods, and handles async Rust. As a result, Paralegal successfully expresses and enforces all these policies.

Policy Structure. Paralegal is designed to distribute work between markers, program analysis, and policy. CodeQL does not have markers (or a marked PDG), and it provides only low-level program analysis primitives. Therefore, we investigated how much of a CodeQL query corresponds to work done by Paralegal outside of the policy. We answered this question by qualitatively labeling each chunk of a CodeQL query (precisely, a CodeQL *predicate*) as either "policy" or "not policy" code. For instance, this CodeQL predicate in the Plume policy to find deletion functions is labeled as "not policy" because it serves the same function as a Paralegal marker:

```
predicate is_delete(DataFlow::Node n) {
  n.asParameter().getFunction().getName()
   .regexpMatch(".*deleteAny.*") and
  n.asParameter().getIndex() = 0
}
```

To check the reliability of this qualitative judgment, two authors of this paper independently labeled 30 predicates and, after resolving one conflict, achieved 96% agreement.

Across all CodeQL policies, we labeled only 36% of predicates as policy code. From our observations, the remaining 64% of code fall roughly into three categories: *(i)* predicates identifying code elements (i.e., markers), *(ii)* predicates defining the semantics of external code (i.e., models), or *(iii)* predicates defining primitives for code analysis. This result shows that Paralegal's design effectively separates concerns that are otherwise mixed together in existing systems.

7.3 Developer Ergonomics

We now evaluate the maintenance effort Paralegal imposes on developers as a codebase evolves. A developer may need to update her Paralegal configuration as follows:

1. she may need to move a marker, e.g., because argument order changes or a function is renamed;
2. she may need to add markers to new code elements;
3. she may need to adjust marker names because of changes to the semantic meaning of a code element; or
4. the privacy engineer may need to change the policy because it made assumptions about application semantics that no longer hold or because the policy goal changed.

Mechanical marker movement is easy, but changes to what markers mean impose higher cognitive developer burden, and policy changes require involving the privacy engineer.

To evaluate this, we run Paralegal on all commits of Atomic, starting with the introduction of the (buggy) permission model, using the same policy and markers as in §7.1. A good result for Paralegal would show that changes to markers are rare and the policy code is robust—i.e., that the policy is abstracted away from code details, and that the markers are at a granularity that avoids frequent changes.

The experiment covers 1,024 commits between June 9, 2021—when Atomic introduced the feature that our policy targets—and March 26, 2024. 87 commits failed to compile

and we skip them. The code Paralegal analyzes for the policy is a subsection of the entire application, but changes over the course of the experiment. We confirmed with the developers that the intent of the policy remains the same throughout. Since all markers assignments are within Atomic's workspace crates, only changes to code in the workspace can require adjustments to markers. On average, Paralegal's analysis touches 907 lines of within-workspace code (min: 411, max: 1,196). Within the 936 functional commits, 66 commits modify one or more analyzed lines, with 60 lines changed on average. Over the entire codebase, Paralegal's analysis touches 22k lines of code affected by 84 commits, with a mean of 1,165 lines changed per commit.

Paralegal detects the permission bug in each commit before the developers' fix (`46a503a`, 186/936 commits in; June 27, 2021) and passes on every commit thereafter. We found two commits that impacted markers: *(i)* a renaming of the marked function (`aba49fe`); and *(ii)* replacing a previously-marked function with one that takes additional configuration arguments (`e0cf2d1`). The former commit required no developer intervention[2] and the second commit caused Paralegal's policy checks to fail. Hence, Paralegal would have alerted the developer even if she neglected to move/adjust the markers previously. The policy itself required no modifications to accommodate code changes—unlike what might be expected if the policy targeted syntactic elements. This indicates that Paralegal's policy/code decoupling is robust.

7.4 Performance and Scalability

We now investigate Paralegal's performance and scalability on the eight case-study applications (Figure 6). We explore three setups that are representative of how we imagine Paralegal will be used in practice: *(i)* running locally on a project's workspace ("Workspace Only"); *(ii)* running interactively on a specific subset of code (e.g., in an IDE plugin); and *(iii)* running across a project's code and its entire dependency tree ("All Dependencies", e.g., in CI). These setups represent different trade-offs in terms of performance and soundness: the first two prioritize low latency, while the third minimizes false negatives as it check properties over all reachable code.

Paralegal is a `rustc` compiler plugin that operators on Rust's MIR. As a consequence, creating PDGs for a target crate requires first compiling all dependencies to make metadata and MIR for each function available. The applications with the largest dependency corpus (Lemmy and Hyperswitch) each take about two minutes to compile. This time is independent of Paralegal configuration and incurred only once, unless dependencies change. Most of this time is in `rustc` and applies to any compiler plugin: at most 4% of time is related to Paralegal-specific tasks, such as persisting the MIR to disk. In the following experiments, we omit this one-off cost and focus on the time Paralegal spends generating the PDGs and checking policies over them.

[2] A renamed marked function retains its marker if developers attach the marker via inline annotation. With markers attached via external configuration, the policy would fail and alert the developer.

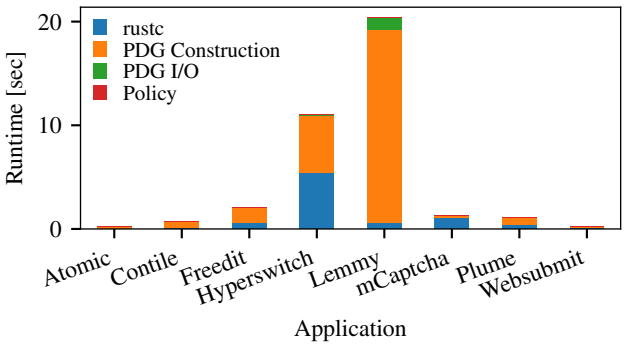


Figure 9: *In the "Workspace Only" setup, which runs Paralegal's analysis only over crates in the current workspace, Paralegal's end-to-end runtime is under 2.2 seconds for most applications. The two exceptions are Lemmy, which has many analysis entrypoints (72) and Hyperswitch, which is a large crate (≈200k LoC). PDG construction dominates runtime.*

We ran all performance experiments ten times. Variance is generally low, with maximum outliers deviating 0.1 seconds from the mean in the "Workspace Only" configuration and 1.2 seconds in the "All Dependencies" configuration.

End-to-End Runtime for "Workspace Only". Most policy-relevant code is usually concentrated in a few crates, such as the workspace a developer is actively working on. In the "Workspace Only" configuration, we set Paralegal to only include code from the current workspace in the PDG and use the modular approximation for functions from other crates. Focusing the analysis on the workspace potentially sacrifices some precision, as it will miss markers assigned in crates outside the workspace, but reduces latency as PDGs are smaller and faster to construct in this setting than when considering all dependencies. Paralegal finds all bugs reported in §7.1, even in this less-precise setup. There are no false positives except in Contile, where adaptive approximation causes a loss of field sensitivity for a helper function, leading to overtaint. Paralegal offers a mitigation strategy for this case, where a developer can configure the analysis to include k^+ more layers of the call tree than adaptive approximation would include in the PDG. In this and all further experiments, we run Contile with $k^+ = 1$, which eliminates the false positive at the cost of 0.3 seconds (9%) additional end-to-end runtime.

We separately measure the time spent in various stages of the analysis: *(i)* `rustc` time outside of the compiler plugin, *(ii)* PDG construction time, *(iii)* time spent (de)serializing PDGs, *(iv)* time spent checking policies over the PDGs. A good result would have Paralegal run in seconds.

Figure 9 shows the results. Most applications finish in under 2.2 seconds, with the exception of Hyperswitch at 12 seconds and Lemmy at 22.5 seconds. This is likely sufficiently fast for

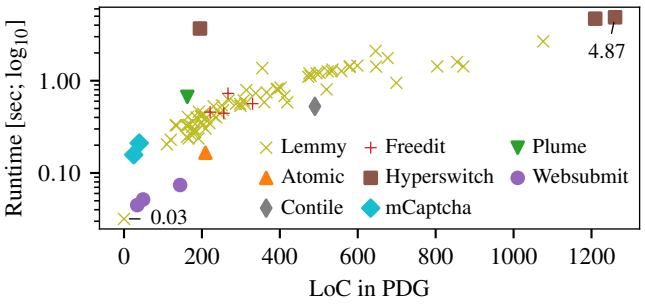


Figure 10: *Paralegal takes a mean of 0.8 seconds per endpoint and around five seconds in the worst case (Hyperswitch), enabling interactive use. Note the* log *scale y-axis.*

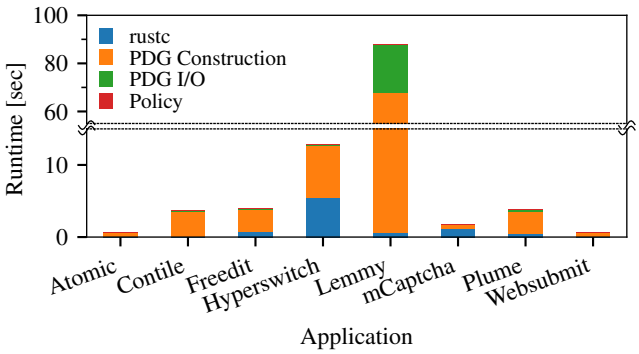


Figure 11: *In the "All Dependencies" setup, which runs Paralegal's analysis over the full codebase (including libraries), Paralegal's end-to-end runtime remains under 20 seconds for all applications except Lemmy (94 seconds), where Paralegal must construct a large PDG for each of the 72 entrypoints.*

interactive CLI or IDE use by a developer.

However, the restriction to workspace crates trades soundness for speed, as a missed marker in a dependency could result in properties spuriously passing (a false negative). While none of our case studies encounter this situation, even simple changes can introduce false-negatives when analyzing only workspace code. For example, Contile's metric data sending logic uses a crate-local error handling function that we marked; if the Contile developers were to switch to an equivalent error handling function provided by a library, Paralegal would miss the marker. Hence, Paralegal deployments might choose to complement quick, local analysis in the "Workspace Only" setup with a CI job that runs the analysis over the full dependency set, as described below ("All Dependencies").

Per-endpoint Runtime. In an interactive setting like an IDE plugin, Paralegal only needs to run on the code that changed at any given time (e.g., the endpoint the developer is editing). We therefore measure the per-endpoint runtime for the "Workspace Only" configuration, and report it as a function of the number of lines of code analyzed. The runtime for each endpoint excludes the overhead of running `rustc` on the target crate, as an IDE setting would use an incremental compiler such as rust-analyzer.

Figure 10 shows the results. Most endpoints take a few seconds (mean: 0.8 seconds), and runtime generally grows with the size of the analyzed code. Hyperswitch has the most expensive endpoints, caused by the policy requiring Paralegal to build a large, slow-to-construct PDG, but remains under five seconds. These results suggest that Paralegal, with incremental compilation, is fast enough for interactive use.

End-to-End Runtime for "All Dependencies". We now consider Paralegal's runtime when analyzing all reachable code, including all dependencies for which source code is available. In our example applications, this results in Paralegal analyzing between 88 and 878 crates (mean: 540). Paralegal still uses the modular approximation for the Rust standard library (`std`, `alloc`, etc.), because the `cargo` package manager uses a precompiled version by default. All runs use the adaptive approximation optimization (§4.2) to reduce the size of the PDG. A good result for Paralegal would show CI-appropriate runtimes of at most several minutes.

Figure 11 shows the results. Most applications finish in less than five seconds and PDG construction dominates runtime in all applications except for mCaptcha. Hyperswitch and mCaptcha have the largest target crates, so `rustc` time is highest for them. Lemmy takes about 94 seconds, as its large number of endpoints (72) results in Paralegal constructing many PDGs (72 PDGs at ≈1.3 seconds each). Overall, the runtime of Paralegal's "All Dependencies" configuration is acceptable for a CI setting.

7.5 Drilldown Experiment

Finally, we measure the impact of Paralegal's adaptive approximation optimization (§4.2).

The optimization determines the smallest prefix of the call tree[3] to be included in the PDG, such that all markers are reached. Paralegal approximates all function calls deeper in the call tree by their type signature. We compare this approach to a setup where the prefix is instead determined via a global depth limit, k. To achieve the same policy accuracy as with adaptive approximation, we assume an (unrealistic) oracle that, for each application, returns the minimal k such that Paralegal reaches all markers. In our setup, the oracle is the maximum depth observed in the adaptively-created PDG. This approach compares Paralegal's adaptive approximation to the strongest possible baseline, since any larger k will result in strictly longer runtime and no improvement in precision. A good result for Paralegal would show that adaptive approximation substantially reduces end-to-end runtime.

In the "Workspace Only" configuration, the gains from adaptive approximation are small, with a 10% mean speedup.

[3]Call graphs can have cycles, but Paralegal breaks cycles via its type-signature-based approximation on recursive calls. This ensures a tree shape.

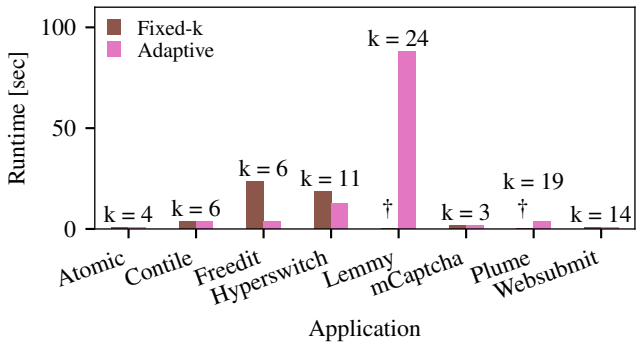


Figure 12: *The adaptive approximation optimization reduces end-to-end runtime by 35% on average and is necessary for Lemmy and Plume to terminate. The label on each bar indicates the oracle-provided minimum k call string length value to match the precision of adaptive approximation. † indicates that the fixed-k version times out after 15 minutes.*

Adaptive approximation makes a much larger difference for the "All Dependencies" configuration. Figure 12 shows the results for this setting. We label each bar with the oracle-provided k value used in the baseline. Most importantly, the adaptive approximation is necessary for PDG construction to terminate in two cases: Lemmy and Plume time out after 15 minutes the fixed-k setting, indicated by a dagger (†) in the figure. For Plume, the buggy version of the application finishes PDG construction in tens of seconds, but the fixed version times out, as it contains more code. In the other applications, the adaptive approximation reduces runtime by 35% on average compared to the idealized fixed-k baseline, with greater gains for applications with larger PDGs.

These results show that adaptive approximation, while always beneficial, is particularly critical when running Paralegal over the full codebase of large applications and their dependencies (e.g., in CI).

8 Related Work

§2 covered closely-related prior work in static analysis tools; we now discuss other approaches that relate to Paralegal.

Information Flow Control (IFC) enforces security policies by attaching security labels to values and propagating them through the program. Static IFC [47, 63, 67] is most comparable to Paralegal, but on its own fails to express important privacy properties (§7.2) and is difficult for developers to adopt [34]. Dynamic IFC can increase precision [13], enforce policies that depend on runtime knowledge, and handle languages which are difficult to analyze statically [76, 80, 82]. However, the tracking incurs runtime overheads which increase with the level of precision used and requires a runtime buy-in. In addition, violation cases lead either to crashes or confusing application behavior in deployment. Paralegal uses static analysis avoid both of these problems.

Bug-finding tools intelligently explore the state space of a program or API to find bugs. Bug finders are easy to adopt and handle a variety of code styles but are usually specialized to a domain such as concurrency [70], distributed systems [75], or file systems [12, 48, 62], and while some offer customization, such as the file-system bug finder eXplode [81], their properties are ultimately hard-coded. Paralegal requires some annotation effort, but allows checking flexible policies while retaining the compatibility with arbitrary code styles.

Code linters identify issues by looking for syntactic patterns such as AST fragments [61] or function names [56]. This approach is simple and practical, but limited in expressiveness and precision. Paralegal uses a more semantic model of the code (a dependency graph) to permit more expressive policies and more robust enforcement.

Policy enforcement tools check policies over data, usually based on runtime mechanisms. The often target domain-specific problems, such as preventing buggy queries [60, 83] or serverless functions [43] from leaking data. Sesame [29] enforces more general policies by leveraging Rust's types and limited static analysis, but requires runtime mechanisms. Paralegal is general and entirely based on static analysis.

9 Conclusion

Paralegal is a practical static analysis tool for checking high-level privacy properties against Rust applications. Paralegal extracts a marked Program Dependence Graph (PDG) from Rust programs annotated with markers, and checks privacy properties written in a high-level DSL against it.

Our evaluation shows that Paralegal finds privacy bugs in real Rust programs, requires modest developer effort as code evolves, and runs in tens of seconds, making it suitable for interactive and CI use. Paralegal is open-source software and available at: `https://github.com/brownsys/paralegal`.

Acknowledgements

We thank Kinan Dak Albab, Nicholas DeMarinis, Franco Solleza, and the members of the ETOS and Systems groups at Brown for their helpful feedback on drafts of this paper. David Fryd contributed several improvements to Paralegal and completed the mCaptcha case study; Tim Nelson helped with an initial version of Paralegal that was based on Forge [39]; and Tim Edgar provided input that inspired the original idea for Paralegal and its auditable, human-readable policy language. Feedback from the anonymous reviewers and Clément Pit-Claudel, our shepherd, greatly improved the paper.

This research was partially supported by NSF awards CNS-2045170, DGE-2208731, and DGE-2335625, by the DARPA under Agreement No. HR00112420354, by a Google Research Scholar Award, a Microsoft Grant for Customer Experience Innovation, an Amazon Research Award, and a gift from VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of our funders.

References

[1] *Actix Web is a powerful, pragmatic, and extremely fast web framework for Rust*. URL: https://actix.rs/ (visited on 04/19/2024).

[2] Justus Adam, Carolyn Zech, Livia Zhu, Sreshtaa Rajesh, Mithi Jethwa, Nathan Harbison, Will Crichton, Shriram Krishnamurthi, and Malte Schwarzkopf. *Paralegal OSDI 2025 Artifact (GitHub)*. May 2025. URL: https://github.com/brownsys/paralegal-osdi-2025-artifact/tree/final.

[3] Justus Adam, Carolyn Zech, Livia Zhu, Sreshtaa Rajesh, Mithi Jethwa, Nathan Harbison, Will Crichton, Shriram Krishnamurthi, and Malte Schwarzkopf. *Paralegal OSDI 2025 Artifact (Zenodo)*. May 2025. URL: https://doi.org/10.5281/zenodo.15374862.

[4] Artem Agvanian. "PEAR: Practical Interprocedural Analyses in Rust". Honors Thesis. Brown University, May 2025.

[5] *Atomic Data is a modular specification for sharing, modifying and modeling graph data*. URL: https://github.com/atomicdata-dev (visited on 04/19/2024).

[6] *AtomicServer (Initial Bug Fix): Prevent Unauthorized Commits*. URL: https://github.com/atomicdata-dev/atomic-server/commit/46a503a (visited on 04/19/2024).

[7] *AtomicServer Privacy Policy*. URL: https://github.com/atomicdata-dev/atomic-server/blob/46a503a/lib/src/commit.rs#L109 (visited on 04/19/2024).

[8] *AtomicServer: a lightweight, yet powerful CMS / Graph Database*. URL: https://github.com/atomicdata-dev/atomic-server (visited on 04/19/2024).

[9] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. "QL: Object-oriented Queries on Relational Data". In: *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP)*. Edited by Shriram Krishnamurthi and Benjamin S. Lerner. Volume 56. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 2:1–2:25. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2016.2.

[10] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. "Efficient and Flexible Discovery of PHP Application Vulnerabilities". In: *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. Paris, France, Apr. 2017, pages 334–349.

[11] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. "Everest: Towards a Verified, Drop-in Replacement of HTTPS". In: *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*. Edited by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Volume 71. Asilomar, California, USA: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, May 2017, 1:1–1:12. URL: https://doi.org/10.4230/LIPIcs.SNAPL.2017.1.

[12] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. "Specifying and Checking File System Crash-Consistency Models". In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, Georgia, USA, Apr. 2016, pages 83–98. URL: https://doi.org/10.1145/2872362.2872406.

[13] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. "HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Vancouver, British Columbia, Canada, Aug. 2015, pages 289–301. URL: https://doi.org/10.1145/2784731.2784758 (visited on 01/02/2023).

[14] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (Sept. 2010). Edited by Andrei Sabelfeld, pages 1157–1210. URL: http://dx.doi.org/10.3233/JCS-2009-0393.

[15] *CodeQL*. URL: https://codeql.github.com/ (visited on 12/03/2024).

[16] CodeQL Developers. *CodeQL C++ library models*. URL: https://github.com/github/codeql/tree/39a67b6e2e6490a9bd010db50e148f647765e9f7/cpp/ql/lib/ext (visited on 12/03/2024).

[17] CodeQL Developers. *CodeQL C++ Semmle models*. URL: https://github.com/github/codeql/tree/39a67b6e2e6490a9bd010db50e148f647765e9f7/cpp/ql/lib/semmle/code/cpp/models/implementations (visited on 12/03/2024).

[18] *CodeQL Documentation: Analyzing data flow in C and C++*. URL: https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-cpp/ (visited on 12/03/2024).

[19] *CodeQL Documentation: Using the guards library in C and C++: The controls predicate*. URL: https://codeql.github.com/docs/codeql-language-guides/using-the-guards-library-in-cpp/ (visited on 12/03/2024).

[20] CodeQL Github Issue #18098. *[C++] How to detect taint on elements in a collection*. URL: https://github.com/github/codeql/issues/18098#issuecomment-2504050120 (visited on 12/03/2024).

[21] CodeQL Github Issue #18099. *[C++] Fails to detect control flow influence of nested "if"*. URL: https://github.com/github/codeql/issues/18099#issuecomment-2500005480 (visited on 12/03/2024).

[22] CodeQL Github Issue #18100. *[C++] Control Flow Influence not detected interprocedurally*. URL: https://github.com/github/codeql/issues/18100 (visited on 12/03/2024).

[23] CodeQL Github Issue #18122. *[C++] Templatized code not found*. URL: https://github.com/github/codeql/issues/18122 (visited on 12/03/2024).

[24] CodeQL Github Issue #18151. *[C++] Taint analysis does not appear to handle aliasing*. URL: https://github.com/github/codeql/issues/18151#issuecomment-2506541314 (visited on 12/03/2024).

[25] CodeQL Github Issue #18171. *[C++] Support for std::async and std::thread*. URL: https://github.com/github/codeql/issues/18171 (visited on 12/03/2024).

[26] *Contile: README.md*. URL: https://github.com/mozilla-services/contile/blob/8170ad2626ddc96a6f74ca794a9c33e9cb76e6e3/README.md#contile-tile-server (visited on 04/19/2024).

[27] *Contile: The back-end server for the Mozilla Tile Service*. URL: https://github.com/mozilla-services/contile (visited on 04/19/2024).

[28] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. "Modular Information Flow through Ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. San Diego, California, USA, June 2022, pages 1–14. URL: https://doi.org/10.1145/3519939.3523445 (visited on 01/09/2023).

[29] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. "Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions". In: *Proceedings of the 30th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Austin, Texas, USA, Nov. 2024, pages 709–725. URL: https://doi.org/10.1145/3694715.3695984.

[30] *Don't allow deleted users to do actions*. URL: https://github.com/LemmyNet/lemmy/commit/2966203 (visited on 04/19/2024).

[31] *Don't allow login if account is banned or deleted*. URL: https://github.com/LemmyNet/lemmy/commit/b78826c2 (visited on 04/19/2024).

[32] *Don't allow posts to deleted / removed communities*. URL: https://github.com/LemmyNet/lemmy/commit/2402515 (visited on 04/19/2024).

[33] Efficient and Trustworty Operating Systems Group @ Brown University. *Paralegal*. July 2025. URL: https://github.com/brownsys/paralegal (visited on 04/16/2024).

[34] Petros Efstathopoulos and Eddie Kohler. "Manageable fine-grained information flow". In: *Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Glasgow, Scotland, UK, Apr. 2008, pages 301–313. URL: https://doi.org/10.1145/1352592.1352624.

[35] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)". In: *Official Journal of the European Union* L119 (May 2016), pages 1–88. URL: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC.

[36] Ming Fan, Le Yu, Sen Chen, Hao Zhou, Xiapu Luo, Shuyue Li, Yang Liu, Jun Liu, and Ting Liu. "An Empirical Evaluation of GDPR Compliance Violations in Android mHealth Apps". In: *Proceedings of the 31st IEEE International Symposium on Software Reliability Engineering (ISSRE)*. Coimbra, Portugal, Oct. 2020, pages 253–264.

[37] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pages 319–349. URL: https://dl.acm.org/doi/10.1145/24039.24041 (visited on 04/16/2023).

[38] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. "RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks". In: *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, California, USA, Dec. 2022, pages 1014–1031. URL: https://www.computer.org/csdl/proceedings-article/sp/2023/933600b014/1Js0DzhaXNm (visited on 01/05/2023).

[39] *Forge: A language built for teaching formal methods and modeling*. URL: https://github.com/tnelson/Forge.

[40] *Freddit: Pageview data retention*. URL: https://github.com/freedit-org/freedit/blob/f5905db9ea3c8630d61c80143d5f2553ee654b15/src/controller/user.rs#L1096 (visited on 04/19/2024).

[41] *Freedit: The safest and lightest forum, powered by rust.* URL: https://github.com/freedit-org/freedit (visited on 04/19/2024).

[42] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. "Information-Flow Analysis of Android Applications in DroidSafe". In: *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*. San Diego, California, USA, Feb. 2015. URL: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/information-flow-analysis-android-applications-droidsafe/ (visited on 04/26/2023).

[43] Praveen Gupta, Arshia Moghimi, Devam Sisodraker, Mohammad Shahrad, and Aastha Mehta. "Growlithe: A Developer-Centric Compliance Tool for Serverless Applications". In: *Proceedings of the 2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, California, USA, May 2025, pages 3161–3179. URL: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00099.

[44] *Hyperswitch: An open source payments switch written in Rust to make payments fast, reliable and affordable*. URL: https://github.com/juspay/hyperswitch (visited on 04/19/2024).

[45] *Introducing mCaptcha net*. URL: https://mcaptcha.org/blog/introducing-mcaptcha-net (visited on 04/19/2024).

[46] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. "Exploring and Enforcing Security Guarantees via Program Dependence Graphs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, Oregon, USA, June 2015, pages 291–302. URL: https://dl.acm.org/doi/10.1145/2737924.2737957 (visited on 04/16/2023).

[47] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. "Cocoon: Static Information Flow Control in Rust". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024). URL: https://doi.org/10.1145/3649817.

[48] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. "Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems". In: *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. Rome, Italy, May 2023, pages 718–733. URL: https://doi.org/10.1145/3552326.3567498.

[49] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. "STORM: Refinement Types for Secure Web Applications". In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, July 2021, pages 441–459. URL: https://www.usenix.org/conference/osdi21/presentation/lehmann (visited on 04/15/2023).

[50] *Lemmy #1656: Deleted users can still make comments and posts*. URL: https://github.com/LemmyNet/lemmy/issues/1656 (visited on 04/19/2024).

[51] *Lemmy #1827: Posting to removed community isn't disallowed*. URL: https://github.com/LemmyNet/lemmy/issues/1827 (visited on 04/19/2024).

[52] *Lemmy #2372: Banned users can act in communities*. URL: https://github.com/LemmyNet/lemmy/issues/2372 (visited on 04/19/2024).

[53] *Lemmy #2372: Deleted account error*. URL: https://github.com/LemmyNet/lemmy/issues/2372 (visited on 04/19/2024).

[54] *Lemmy #2372: Deleted users can act in communities*. URL: https://github.com/LemmyNet/lemmy/issues/2372 (visited on 04/19/2024).

[55] *Lemmy: A link aggregator and forum for the fediverse*. URL: https://github.com/LemmyNet/lemmy (visited on 04/19/2024).

[56] Baptiste Lepers, Josselin Giet, Willy Zwaenepoel, and Julia Lawall. "OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel". In: *Proceedings of the 18th European Conference on Computer Systems*. Rome, Italy, May 2023, pages 33–45. URL: https://doi.org/10.1145/3552326.3567504.

[57] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. "Mining Node.js Vulnerabilities via Object Dependence Graph and Query". In: *Proceedings of the 31st USENIX Security Symposium*. Boston, Massachusetts, USA, Aug. 2022, pages 143–160. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/li-song.

[58] *mCaptcha #161: Privacy Bug: Over-collection in Opt-in Performance Statistics*. URL: https://github.com/mCaptcha/mCaptcha/issues/161.

[59] *mCaptcha: Proof of work based, privacy respecting CAPTCHA system*. URL: https://github.com/mCaptcha/mCaptcha (visited on 04/19/2024).

[60] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. "Qapla: Policy compliance for database-backed systems". In: *Proceedings of the 26th USENIX Security Symposium*. USENIX Security '17. Vancouver, British Columbia, Canada, Aug. 2017, pages 1463–1479. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mehta.

[61] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Transactions on Software Engineering* 36.1 (Jan. 2010), pages 20–36.

[62] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. "Finding crash-consistency bugs with bounded black-box crash testing". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018, pages 33–50.

[63] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. San Antonio, Texas, USA, Jan. 1999, pages 228–241. URL: https://doi.org/10.1145/292540.292561 (visited on 02/02/2023).

[64] *Plume #1144: delete comments when deleting users*. URL: https://github.com/Plume-org/Plume/commit/19f18421bcd9cb9d1654de24f9a04747691036b7 (visited on 04/16/2023).

[65] *Plume #806: Media not deleted after account deletion*. URL: https://github.com/Plume-org/Plume/issues/806 (visited on 04/19/2024).

[66] *Plume: A federated blogging engine based on ActivityPub*. URL: https://github.com/Plume-org/Plume (visited on 04/19/2024).

[67] François Pottier and Vincent Simonet. "Information flow inference for ML". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Portland, Oregon, USA, Jan. 2002, pages 319–330. URL: https://doi.org/10.1145/503272.503302.

[68] Malte Schwarzkopf. *Websubmit-rs: A Simple Class Submission System*. URL: https://github.com/ms705/websubmit-rs (visited on 04/06/2022).

[69] Oliver Smith. *The GDPR Racket: Who's Making Money From This $9bn Business Shakedown*. URL: https://www.forbes.com/sites/oliversmith/2018/05/02/the-gdpr-racket-whos-making-money-from-this-9bn-business-shakedown/ (visited on 02/01/2023).

[70] Bogdan Alexandru Stoica, Shan Lu, Madanlal Musuvathi, and Suman Nath. "WAFFLE: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection". In: *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. Rome, Italy, May 2023, pages 111–126. URL: https://doi.org/10.1145/3552326.3567507.

[71] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. "If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems". In: *Proceedings of the 30th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Austin, Texas, USA, Nov. 2024, pages 63–78. URL: https://doi.org/10.1145/3694715.3695971.

[72] *The Cost of Continuous Compliance*. Feb. 2020. URL: https://www.datagrail.io/resources/reports/gdpr-ccpa-cost-report/ (visited on 02/01/2023).

[73] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. "Securify: Practical Security Analysis of Smart Contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Toronto, Canada, Oct. 2018, pages 67–82. URL: https://doi.org/10.1145/3243734.3243780.

[74] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. "Verifying Dynamic Trait Objects in Rust". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Pittsburgh, Pennsylvania, USA, Oct. 2022, pages 321–330. URL: https://doi.org/10.1145/3510457.3513031.

[75] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. "Model Checking Guided Testing for Distributed Systems". In: *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. Rome, Italy, May 2023, pages 127–143. URL: https://doi.org/10.1145/3552326.3587442.

[76] Frank Wang, Ronny Ko, and James Mickens. "Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services". In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Feb. 2019, pages 615–630. URL:

https://www.usenix.org/conference/nsdi19/presentation/wang-frank (visited on 04/07/2022).

[77] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. "PrivGuard: Privacy Regulation Compliance Made Easier". In: *Proceedings of the 31st USENIX Security Symposium*. Boston, Massachusetts, USA, Aug. 2022, pages 3753–3770. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-lun (visited on 12/29/2022).

[78] John Whaley and Monica S. Lam. "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*. Washington DC, USA, June 2004, pages 131–144. URL: https://doi.org/10.1145/996841.996859.

[79] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. San Jose, California, USA, May 2014, pages 590–604. URL: https://ieeexplore.ieee.org/abstract/document/6956589.

[80] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, dynamic information flow for database-backed applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Santa Barbara, California, USA, June 2016, pages 631–647. URL: https://doi.org/10.1145/2908080.2908098.

[81] Junfeng Yang, Can Sar, and Dawson Engler. "Explode: a lightweight, general system for finding serious storage system errors". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006, pages 131–146. URL: https://www.usenix.org/legacy/event/osdi06/tech/full_papers/yang_junfeng/yang_junfeng.pdf.

[82] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Improving Application Security with Data Flow Assertions". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pages 291–304. URL: https://doi.org/10.1145/1629575.1629604.

[83] Wen Zhang, Eric Sheng, Michael Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. "Blockaid: Data Access Policy Enforcement for Web Applications". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, July 2022, pages 701–718. URL: https://www.usenix.org/conference/osdi22/presentation/zhang.

[84] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. "Automated Analysis of Privacy Requirements for Mobile Apps". In: *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*. San Diego, California, USA, Feb. 2017. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_05A-5_Zimmeck_paper.pdf.

A Appendix

A.1 Case Studies In Depth

(1) Atomic is a graph database server that lets users create, edit, and share graph-structured data [5, 8]. It uses the Actix web framework [1].

Each time that a user modifies a database resource, Atomic stores a signed *commit* record. Before creating a commit, the application must verify that the user has permissions to modify that resource. Earlier "parent" commits for the resource specify permissions for later commits. For instance, a commit that creates a new resource must also specify which users are allowed to edit that resource. For each commit, the application must check the permissions on the commit's parent. Crucially, this authorization must happen *before* updating the in-memory copy of the database resource [7]. If the authorization check happened *after* applying the commit, the user could first change their parent commit to one that gives them the requisite permissions, thereby guaranteeing that they pass the permission check.

To encode this policy in Paralegal, we define a `commit` marker for the `Commit` type. We then create a `modify_resource` marker for all operations that modify in-memory references to resources. We add a `sink` marker to the write operation that flushes the modified resource back to the database. Finally, we add an `auth_check` marker for checking the user's permissions. The Paralegal policy enforces that if a `resource` flows to a `modify_resource` then to a `sink`, the `resource` flows to an `auth_check`, and there is a control flow influence from the `auth_check` to the `modify_resource`.

(2) Contile is the backing server of the Mozilla Tile Service, which serves as an intermediary between advertisers and the landing page of the Firefox web browser to "ensure customer privacy" in the advertising context [26]. Contile services small ads, called *tiles*, to Firefox users.

The information available to the application is centered around the `Tags` struct, which features the following comment referencing the fields `tags` and `extra` of the struct:

> Not all tags are distributed out. ‘tags‘ are searchable and may cause cardinality issues. ‘extra‘ are not searchable, but may not be sent to [Metrics].

The Paralegal policy ensures that the data from the `extra` field is withheld from advertiser search and that it is never sent to the metrics server. We marked the `extra` field as sensitive data. We then marked the sinks to which `extra` should not flow: the first argument of both `MetricsBuilder::try_send` from the `cadence` library and `RequestBuilder::send` from the `reqwest` HTTP library.

(3) Freedit is a forum application that supports both Twitter and Reddit-like interaction modes [41]. Freedit's code comments state that it only retains a user's viewing history for three days [40].

We write two Paralegal policies. The first requires that when page view data is stored, a timestamp for its expiration date is stored alongside it. The second requires that there exists code that checks the database for expired page view data and deletes it. We marked the identifier for the table that stores pageview data for a user. We also marked the library functions that insert and delete data from the data store, as well as a library function for getting the current time.

(4) Hyperswitch is a payment router that provides a unified interface for interacting with common payment processors.

Hyperswitch's UI asks users to opt-in to saving their credit card information for future transactions. We write a policy that mandates the user's selection determines (via control flow) whether the credit card details are indeed stored. Hyperswitch's documentation also states that plain-text API keys are only available at creation. We write a Paralegal policy that states that an API key may only be released once to the user creating it, and that it can only be access through its hash afterwards.

We marked the type identifying credit card details and the function returning the user's storage decision. Additionally, we marked the API key type, the permissible hash function, and the response type for the key's initial creation. We also marked the return values of all endpoints to ensure that Paralegal captures all data sent to users. We analyze the the controller that creates API keys and two controllers that handle credit card data.

(5) Lemmy is a federated Reddit-like platform [55]. Rather than providing a single centralized website, anyone can create an instance tailored to their interests and moderation preferences. Users create communities within those instances and post content to them.

Our Lemmy policies focus on its access control rules: *(i)* if a user is banned or deleted from an instance, they may not read nor write data in that instance; and *(ii)* if a user is banned from a community or the community is permanently removed, users may not write data to that community.

We define four authorization check markers: one for an instance ban check, one for an instance deletion check, and two more for the respective community checks. We define `instance` and `community` markers for database accesses pertaining to data relevant to instances or communities respectively. Authorization checks need to happen in 72 HTTP endpoints in Lemmy that perform database reads and writes. Our policy stipulates that respective authorization checks need to take place in each controller where instance or community database accesses take place and that such checks must happen before the access and have a control flow influence. In addition to the bugs we report, we also found two more controllers that violate this policy. The controllers in question create new sites or communities. In this special circumstance, a limited number of database accesses are performed before the checks to complete the initial setup of the new site or community. We manually verified that performing these accesses is safe and then marked these locations as exceptions from the policy.

(6) mCaptcha is a proof-of-work based CAPTCHA service focused on privacy [59]. Website developers register their sites with mCaptcha and invoke the mCaptcha service API when end-users visit those sites. If a developer deletes their mCaptcha account, mCaptcha must remove all data associated with it. We realize this with a policy similar to §3's example and mark the `Identity` type from `actix_identity` as well as the `delete_user` method.

mCaptcha's PoW based algorithm features a tunable "difficulty factor", designed to balance security and accessibility to legitimate users. To optimize this parameter, mCaptcha can share statistics with other mCaptcha installations. Publishing this data requires explicit developer opt-in [45]. We initially wrote a policy to enforce that performance data could only be collected from websites that opted in. This verify-before-collect policy failed, and after discussing the issue mCaptcha developers, it turned out that we misinterpreted their privacy goals: statistics are always collected, even if developers don't opt into sharing them. A corrected verify-before-sharing policy passes Paralegal. However, the discussion helped the mCaptcha developers find a (related) data integrity bug: they deleted statistics when a user revoked their opt-in consent, even though they didn't intend to [58]. This illustrates that Paralegal-induced discussions can be helpful to developers.

(7) Plume. Plume is a federated blogging service [66], and the basis of our example in §3. If a user deletes their Plume account, the application must delete their personally identifiable data. We formalize this policy with Paralegal by marking the user and the types of user data: `Comment`, `Blog`, `Post`, `Media`, and `Notification`. Additionally, we placed a marker on the `delete` function in the `diesel` database ORM, for a total of seven markers.

(8) WebSubmit [68] is a homework submission system deployed at a U.S. university and written in 1.6k LoC of Rust using Rocket.rs. We consider three policies: *(i) data deletion*, which tests compliance with a GDPR-style "right to be forgot-

ten" by ensuring that an endpoint for deleting all of a user's data exists; *(ii) scoped storage*, which ensures that the user's identity is stored alongside their data; and *(iii) authorized disclosure*, which encodes the access-control policy: students may view their own answers, TAs and instructors may view all students' answers, and instructors may view course feedback.

We marked the data type containing student answers, deletion functions, the return value of each controller (to cover externalizing data), user identifiers provided by the framework as well as functions that retrieve instructors and TA's from the config.

The analysis covers the endpoint that stores student submitted answers and the deletion controller.

A.2 Source Code Changes

The following table lists, for each application, how many no-op functions we had to introduce to attach markers and how often they needed to be called.

Application	# Marker Functions (calls)
Contile	2 (5)
Freedit	1 (4)
mCaptcha	1 (1)
Hyperswitch	1 (1)
WebSubmit	2 (2)

In addition, we had to make the following adjustments to make analysis feasible or work around prototype limitations:

- **Contile**: We inlined one call to `Result::map_err`. This is a small function critical to the policy, but because it is in the (precompiled) standard library our multi-crate analysis could not access its code.
- **Hyperswitch**: We changed the type `PlaintextApiKey` to use an explicit `prefix` and `key` fields instead of a single string and a marked accessor function for the sensitive `key` field.
- **Lemmy**: We created (generalized) model for `actix::web::block` to ensure the closure it receives can be analyzed without having to deal with the `unsafe` code of `block`.
- **mCaptcha**: We stubbed compile time generated code by the "cachebust" utility as it failed to run on our machine and is irrelevant to the policy.
- **Freedit**: We moved the `user_cron_job` function (11 LoC) from the binary to the library. This works around a limitation in the `Somewhere:` policy scope, which can currently only see the entry points from one crate at a time and was causing a false positive.

We also made the following changes to be able to express a cleaner policy, as described in §6:

- **Atomic**: We factored a benign modification of the parent of a commit into a helper function that we marked.
- **Freedit**: We factored a comparison used for checking expired data into a helper function that is explicitly marked.
- **Lemmy**: We added calls to a `policy_exception` function that exempts a limited number of accesses from the policy check as described in Appendix A.1.

A.3 Policy DSL Grammar

⟨*paralegal policy*⟩ ::= `Scope:` ⟨*scope*⟩
`[ Definitions:` ⟨*definitions*⟩ `]`
`Policy:` ⟨*exprs*⟩

⟨*definitions*⟩ ::= ⟨*definition*⟩ ⟨*definitions*⟩ | ⟨*definition*⟩

⟨*definition*⟩ ::= ⟨*bullet*⟩ ⟨*variable*⟩ `is each` ⟨*variable_intro*⟩
`where:` (⟨*exprs*⟩ | ⟨*body*⟩)

⟨*scope*⟩ ::= `Everywhere:` | `Somewhere:` | `In` ⟨*controller*⟩`:`

⟨*exprs*⟩ ::= ⟨*clause*⟩ ⟨*operator*⟩ ⟨*exprs*⟩
| ⟨*clause*⟩
| ⟨*only via relations*⟩

⟨*bullet*⟩ ::= ⟨*number*⟩`.` | (⟨*number*⟩`.`) | ⟨*letter*⟩`.` | (⟨*letter*⟩`.`)

⟨*operator*⟩ ::= `and` | `or`

⟨*clause*⟩ ::= ⟨*clause intro*⟩ ⟨*clause body*⟩

⟨*clause intro*⟩ ::= ⟨*for each*⟩ | ⟨*there is*⟩

⟨*for each*⟩ ::= ⟨*bullet*⟩ `For each` ⟨*variable intro*⟩`:`

⟨*there is*⟩ ::= ⟨*bullet*⟩ `There is a` ⟨*variable intro*⟩ `where:`

⟨*variable intro*⟩ ::= ⟨*variable*⟩ `input`
| ⟨*variable*⟩ `item`
| ⟨*variable*⟩ `type marked` ⟨*marker*⟩
| ⟨*variable*⟩ `that produces` ⟨*variable*⟩
| ⟨*constrained variable*⟩

⟨*clause body*⟩ ::= (⟨*clause*⟩ | ⟨*body*⟩) ⟨*operator*⟩ ⟨*clause body*⟩
| ⟨*clause*⟩
| ⟨*body*⟩

⟨*body*⟩ ::= ⟨*bullet*⟩ ⟨*relation*⟩ ⟨*operator*⟩ ⟨*body*⟩
| ⟨*conditional*⟩
| ⟨*bullet*⟩ ⟨*relation*⟩

⟨*conditional*⟩ ::= ⟨*bullet*⟩ `If` ⟨*relation*⟩ `then:` ⟨*clause body*⟩

⟨*only via relations*⟩ ::= ⟨*only via relation*⟩
| ⟨*only via relation*⟩ ⟨*operator*⟩ ⟨*only via relations*⟩

⟨*constrained variable*⟩ ::= ⟨*variable*⟩
| ⟨*variable*⟩ `marked` ⟨*marker*⟩

⟨*only via relation*⟩ ::= `Each` ⟨*variable intro*⟩
`goes to a` ⟨*constrained variable*⟩
`only via a` ⟨*constrained varaiable*⟩
`marked` ⟨*marker*⟩

⟨*relation*⟩ ::= ⟨*variable*⟩ `does not influence` ⟨*variable*⟩
| ⟨*variable*⟩ `influences` ⟨*variable*⟩
| ⟨*variable*⟩ `goes to` ⟨*variable*⟩
| ⟨*variable*⟩ `does not go to` ⟨*variable*⟩
| ⟨*variable*⟩ `affects whether` ⟨*variable*⟩ `happens`
| ⟨*variable*⟩ `does not affect whether` ⟨*variable*⟩ `happens`
| ⟨*variable*⟩ `is marked` ⟨*marker*⟩
| ⟨*variable*⟩ `is not marked` ⟨*marker*⟩

B Artifact Appendix

Abstract

Our artifact contains the code for the Paralegal analyzer, policy compiler, supporting libraries, the set of applications we evaluate on, the benchmarker used to run evaluations, and the plotting script to generate the figures in the paper.

Scope

The artifact allows reproducing the following results:

1. Comparison with CodeQL (column three in Figure 8).
2. Policy structure and coding claims (paragraph "Policy Structure" in §7.2.2).
3. Bugs found by Paralegal (§7.1).
4. The ergonomics report from §7.3.
5. All claims made in performance evaluations in §7.4, such as Figures 9 to 12.

Contents

The artifact contains:

1. Source code for the Paralegal code analyzer, policy compiler, and supporting libraries for annotations and the low-level Rust API for policies.
2. The source code for all case study applications mentioned in the paper, as well as their Paralegal policies.
3. C++ translations for the case study applications and the CodeQL policies we tried on them.
4. Source code for the benchmarker used to run the evaluations.
5. The plotting script used to generate the figures in the paper.

The Docker container version of the artifact additionally contains the CodeQL version we used and all dependencies needed for any software that runs or is analyzed during the benchmarks.

Hosting

We provide two versions of the artifact: a Docker image hosted on Zenodo [3] and a GitHub repository [2] (tag: `final`).

The container is intended as an everything-included solution that allows for validation without any (online) dependencies.

The GitHub repository contains only the immediate source code for the programs and use cases, as well as instructions and lockfiles to replicate the environment used for the evaluation.

Requirements

Please refer to the README distributed with the artifact for detailed explanations of the requirements for both artifact versions. The Docker container version requires an x86_64 machine with Docker.

B.1 Installation and Experiment Workflow

We provide detailed instructions for installation, running experiments, and interpreting results in the README files for both the Docker/Zenodo and GitHub versions of the artifact.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

MettEagle: Costs and Benefits of Implementing Containers on Microkernels

Till Miemietz[†] Viktor Reusch[†] Matthias Hille[†] Lars Wrenger[*] Jana Eisoldt[†] Jan Klötzke[‡]
Max Kurze[♯] Adam Lackorzynski[‡♯] Michael Roitzsch[†] Hermann Härtig[†♯]

[†]*Barkhausen Institut, Germany* [*]*Leibniz-Universität Hannover, Germany*
[‡]*Kernkonzept GmbH, Germany* [♯]*Technische Universität Dresden, Germany*

Abstract

Today, many applications are hosted by cloud providers. In order to isolate the workloads of different clients, cloud enterprises mostly rely on containers rather than standard processes, since the latter are able to exercise a lot of ambient authority. Containers counter this deficiency by sandboxing processes. To this end, they use dedicated security mechanisms such as `seccomp-bpf`. However, these mechanisms add complexity to the kernel and increase its attack surface, thus prompting new security challenges.

Processes in microkernel-based systems do not have ambient authority. Thus, they do not require additional security mechanisms to build sandboxes. In this paper, we try to answer the question whether a microkernel-based OS architecture enables a leaner and more secure container infrastructure. Based on a CVE analysis, we show that the conceptual simplicity of containers on microkernels results in a better security posture than that typically found on monolithic systems.

We furthermore demonstrate the practical feasibility of implementing containers on state-of-the-art microkernels by building MettEagle, a prototype container service running on L4Re. We found that applications running in containers on L4Re expose performance characteristics comparable to that of containers on Linux for both synthetic and real-world benchmarks. In some cases, the container implementation of L4Re even outperforms Linux, accelerating container startup latency and improving network performance.

1 Introduction

Currently, virtual machines (VMs) and containers are the two prevalent mechanisms for isolating untrusted workloads running on a shared system. Generally speaking, containers are a more lightweight approach to isolation as they do not attempt to emulate an entire computing platform. Instead of spawning distinct operating system instances, multiple containers running on the same host share a single kernel and only expose separate instances of userland to applications. This allows them to achieve runtime performance close to that of the bare metal system [48]. Moreover, starting a container is much faster than booting a virtual machine [42]. These advantages made containers the premier choice for isolating workloads in settings such as serverless computing [38].

Interestingly, when looking at containers from the perspective of an OS engineer, they are only little more than hardened processes (or process groups) that are restricted in their resource utilization through dedicated in-kernel mechanisms such as `cgroups` or `seccomp-bpf` [1]. These facilities however, add a lot of complexity to core OS abstractions. This additional complexity in turn harms cross-container isolation by exposing a larger in-kernel code base that is shared between distrusting containers [4, 5].

In fact, the intricacies of implementing an isolation mechanism equivalent to that of containers on traditional, monolithic OSes largely stems from the fact that these architectures do not adhere to the principle of least authority (PoLA) by design. Hence, on such systems, the implementation of containers needs to prevent applications from exercising ambient authority that they should not have in the first place. One example of this problem is the use of `seccomp-bpf` to retroactively restrict the set of system calls (and their parameters) that is available to a container. This does not only add complexity to the kernel but also introduces a slight performance penalty since the respective checks need to be carried out at runtime [34]. Even though this overhead has been reduced recently [2], it will never disappear completely.

Modern microkernel-based OS designs [35, 37], in contrast, enforce PoLA by default. On such systems, processes have to explicitly request access to any system service like drivers or file systems instead of implicitly having the authority to use them. As a consequence, implementing container-style isolation on such platforms does not require additional resource constraining mechanisms as the OS itself provides strong compartmentalization of processes by default. Hence we argue, that on a microkernel-based OS, the use of processes can provide better isolation properties than containers on monolithic systems.

From a security perspective, a microkernel design can signif-

icantly decrease the amount of code that mutually distrusting containers need to share since the core system that each process has to rely on is small. Moreover, as all other system services are implemented as separate processes in userspace, the trusted computing base (TCB) of a container running on a microkernel only includes services that it actually uses [30]. This is different from monolithic approaches where kernel parts that are only used by some containers contribute to the TCB of all clients running on the same machine. Consequently, microkernels enable container implementations with an improved security posture.

However today, microkernel-based OS architectures are mostly used in embedded systems with statically configured workloads. First, this raises the question of how well microkernel concepts scale to large hardware platforms – the type of hardware that containers are usually deployed on. Second, to the best of our knowledge, the performance implications of using capability-based microkernels to host highly dynamic workloads like serverless computing services have not been studied so far.

In order to answer these questions and investigate the cost and benefits of our concept for containers on microkernels, we built a prototype container engine on top of L4Re [27], a state-of-the-art microkernel-based OS. We show how to provide container-grade isolation, including resource restriction and the necessary system services on such a system. Following a detailed discussion on the security properties of the resulting approach, we analyze its performance for both microbenchmarks as well as in a function-as-a-service (FaaS) setting, using the SeBS benchmark suite [16] as an evaluation platform.

In summary, this paper makes the following contributions:

- An analysis of what is needed to provide container-grade isolation to applications (Section 2).
- The design (Section 3) and implementation (Section 4) of a container-style isolation mechanism on L4Re, a microkernel-based OS. This includes modifications of L4Re to improve its performance when facing dynamic workloads on server-grade hardware.
- A detailed evaluation of our approach from both a security perspective (Section 5) as well as from a performance angle (Section 6), with a particular focus on a comparison to standard frameworks for lightweight virtualization on Linux, namely runC and Firecracker.

2 Background

Containers or lightweight virtual machines are used for isolating workloads in cloud environments. In the following we will discuss their general properties and implementation on monolithic OSes. Afterwards, we introduce L4Re and its capability-based access-control mechanism which we leverage for our secure container architecture.

2.1 Containers on a Monolithic OS

Broadly speaking, a container is an OS mechanism that virtualizes the execution environment of applications while sharing a single kernel. Examples for such facilities can be found on a variety of operating systems such as BSD (*Jails* [33]), Solaris (*Zones* [52]), and Linux [22]. Even though the exact implementation differs, containers on all platforms rely on three common kernel mechanisms with respect to security: 1. Deny access to unnecessary (kernel) interfaces. 2. For necessary interfaces, the visibility of resources is restricted. 3. Where resources have to be shared, resource accounting is enforced.

Interface Restrictions: As shown by Canella et al. [14], many cloud applications indeed only require a fraction of the rather large kernel interface exposed by monolithic OS designs. Container implementations exploit this fact to increase security by preventing a container from executing certain system calls, thus reducing the attack surface of the shared kernel. Current solutions like `seccomp-bpf` on Linux are built on top of the Berkeley Packet Filter (BPF) [43] and allow administrators to specify small filtering programs that the kernel executes upon every system call of a container to determine whether it adheres to the container's security policy.

Visibility Restrictions: Containers typically impose additional visibility restrictions on top of traditional processes. By using kernel features like Linux' `namespaces`, the administrator is able to hide certain parts of the system from applications running inside a container.

Within the scope of containers, visibility restrictions are applied e.g. to the file system by setting a separate root directory using the `chroot` system call. Container implementations often use visibility restriction mechanisms to virtualize network interfaces, process IDs, and mount points. Also, visibility restriction measures assure that processes running inside a container cannot communicate across container boundaries.

Resource Restrictions: For reasons of performance isolation, operating systems also restrict the resource usage of containers using dedicated mechanisms. One prominent example of such resource constraining frameworks is Linux' `cgroups` feature. Resource restriction frameworks for containers are often capable of organizing restrictions in a hierarchical manner, allowing containers to pass on a fraction of their already constrained resource budget [10]. Furthermore, they can prioritize access to system resources, thus guaranteeing a minimum share of a certain resource to each container.

Beyond these three core mechanisms, most implementations of containers also provide an ecosystem for managing containerized applications. For instance on Linux, a *high-level container runtime* like containerd [24] manages container images and the resources that a container requests for

execution such as access to the network. The Open Container Initiative (OCI) provides standards for the container runtime interface and container images, which allows OCI-compliant container managers to interact with any OCI-compliant *low-level container runtime*. Low-level container runtimes like runC [32] are responsible for setting up a container by configuring the lightweight virtualization mechanisms provided by the OS. After a container is launched by the low-level container runtime, the container manager takes over monitoring and life cycle management of the container.

2.2 Lightweight Virtual Machines

Virtual machines (VMs) provide a higher degree of isolation than containers, as distrusting applications do not share a kernel instance. Lightweight virtual machine implementations like Firecracker [7] and Cloud Hypervisor [6] try to overcome traditional performance shortcomings of VMs by optimizing virtual machine monitors for use cases like serverless computing, thus providing performance close to that of containers [9]. For instance, reducing the hardware emulated by the hypervisor is one way how Firecracker optimized its boot time.

The use of unikernels as a VM guest [42] is another way to increase the performance of VMs, as unikernels run in a single privilege level, saving context switches during execution. Furthermore, a unikernel can strip unused parts from the OS and userspace components, reducing the size of the TCB [54]. However, the missing isolation between the OS components and the application inside a unikernel contradicts the objective of a multi-layered security approach.

2.3 A Primer on the L4Re Microkernel OS

Being a microkernel, the L4Re kernel itself only implements functionality that cannot be realized in userspace, such as page table manipulation or mechanisms for inter-process communication (IPC). The L4Re microkernel (also known as the `Fiasco` kernel) represents each of the abstractions that it offers to userspace processes as *kernel objects*. Prominent examples for kernel objects are threads, IRQs, the scheduler and *IPC gates* which represent an IPC channel to a process.

Every other component of the operating system such as the memory management or device drivers run as processes in userspace (dubbed *tasks* in L4Re). During bootup the L4Re microkernel starts a *root task* called `moe`. `Moe` provides the basic system abstractions of L4Re such as RAM allocation or access to the read-only boot file system to other applications. `Moe` allows defining quotas for resource allocations, thus being able to constrain the resource consumption of application tasks. After initialization, `moe` starts `ned`, a service task that loads actual L4Re applications as specified by a startup script which is a part of an L4Re boot image. The components of L4Re we base our system on have single-threaded implementations.

2.4 Capabilities

L4Re, like most modern microkernel-based operating systems, uses capabilities [18] for implementing access control. A capability is an unforgeable token of authority, that grants the possessing process the power of carrying out operations on objects. For instance such an operation could be an IPC call to another process.

A central feature of capability-based access control is the absence of ambient authority. Tasks in L4Re start with no authority, i.e., no capabilities by default. This is crucial for a secure container infrastructure. To perform useful work and interact with the system, tasks must be granted the required capabilities explicitly. Other parts of the system cannot be accessed and are in fact not even visible to said task. Sharing object access is simple, as the owner of a capability is free to grant it to other tasks (*delegation*) with equal or less *rights* (for example, restricting read/write to read). Contrary to delegation, the owner of a capability can *revoke* (i.e., void) it, preventing its future use for authorization.

In L4Re, capabilities manifest as the permission to interact with a certain kernel object. Depending on the type of the respective object and the rights of the capability pointing to it, such interactions may configure the CPU scheduler, send an IPC message to another thread via an IPC gate, or destroy a kernel object. With L4Re, a capability can be delegated by sending it over an IPC gate connected to another process, whereupon the kernel takes care of copying the capability into the target process of the IPC. Similarly, the owner of a capability can instruct the kernel to revoke it at any time. Consequently, the L4Re microkernel is responsible for maintaining the system's security by shielding the capabilities of all processes against unauthorized manipulation.

3 Containers on L4Re

This section describes how the functional features and the security properties of Linux containers can be implemented on a microkernel-based OS like L4Re. In the following, we will use the term *compartment* for referring to an entity with container-style isolation running on L4Re, differentiating it from the Linux equivalent which we keep calling *container*. We start with an overview of our architecture and exemplify the interaction of the components by walking through the life cycle of a compartment. Finally, we explain how the isolation mechanisms used by containers on Linux can be transcribed to a microkernel-based OS, and whether these mechanisms already exist in L4Re, need to be introduced, or can be omitted entirely.

3.1 L4Re Compartment Architecture

Figure 1 depicts the components of our microkernel-based compartment system, which runs on top of the L4Re base system. This base system contains the L4Re microkernel and

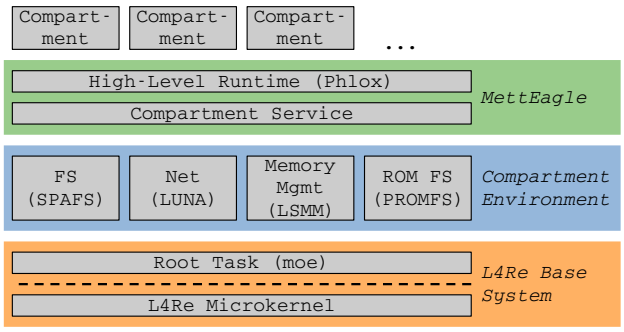


Figure 1: A sketch of L4Re's compartment architecture. Each gray box represents an L4Re process (*task*).

the root task `moe` which manages initial memory allocations to our services at system boot and provides other L4Re system primitives. The *compartment environment* contains all services required to run applications inside L4Re compartments. This includes services for file systems, networking, and scalable memory management.

The management of compartments on L4Re is handled by a *compartment engine* called *MettEagle*. Similar to the container stack on Linux, MettEagle comprises two subservices. The *compartment service* sets up the OS mechanisms required for running a compartment and manages the tasks associated with it. In Linux terminology, the compartment service is similar to a low-level container runtime like runC.

The compartment service is accompanied by a high-level runtime called *Phlox*. Phlox provides a more abstract interface to the compartment engine and implements a FaaS API that allows to spawn compartments remotely via network. For simplicity, we use the term *compartment engine* to refer to the service pair consisting of both the compartment service and the high-level runtime.

As common for microkernel architectures, each of the services depicted in Figure 1 runs in a separate task, yielding strong isolation between them as well as modularity. Note however, that a microkernel design does neither reduce the complexity of these services nor the effort of implementing them.

3.2 Life Cycle of a Compartment

Upon receiving a request for launching a compartment, MettEagle first gathers the resources that the respective compartment needs to run. In detail this means that Phlox creates new sessions with each of the system services in charge of managing the requested resources. For instance, this could be a network or a file system service. As a result of creating a new session with a system service, Phlox receives a capability to a new IPC gate via which the compartment is going to access the system service.

Now, Phlox instructs the compartment service to launch the compartment. Along with this request, Phlox sends the compartment service the set of capabilities gathered during the resource allocation phase. As shown in Figure 2, the compartment service delegates this set of capabilities to each task of the new compartment. These initial capabilities allow the tasks of a compartment to access the system services required for running an application. Now, the compartment service launches the compartment's tasks and monitors their exit status.

As the last step in the life cycle of a compartment, the compartment service collects the artifacts left over from the execution of a compartment. This includes closing all remaining sessions allocated for a compartment by revoking the corresponding capabilities. This in turn triggers the deletion of the associated kernel objects and frees the resources attached to them. MettEagle finally reports the compartment's exit status and optionally statistics on the compute time consumed by a compartment.

3.3 Isolating Compartments

With the general operation of MettEagle clarified, we now detail how microkernel-based compartments differ from Linux' containers.

Visibility Restrictions: Just as with Linux, at the core, a compartment on L4Re consists of a set of processes. All tasks of a compartment share a set of capabilities, hereinafter called the *compartment's capabilities*. These capabilities define which parts of the system are visible to the tasks of the compartment. Thus, by carefully choosing which capabilities are delegated to a compartment, MettEagle can restrict the system surface exposed to a compartment in a manner similar to Linux' `namespaces`. MettEagle can for instance constrain the set of processes a compartment can communicate with, or the network interfaces that are accessible to a compartment.

Another cornerstone for providing visibility restrictions in MettEagle is the absence of globally shared resources and identifiers on L4Re. For example, L4Re has no notion of PIDs or shared memory keys. Instead in L4Re, the respective objects are referred to by capabilities. As a consequence, MettEagle does not need to implement the respective virtualization layers found in visibility restriction mechanisms on monolithic kernels.

In order for an application to easily discover the parts of the system accessible to its compartment, we leverage L4Re's `namespaces` that allow a task to refer to a capability using a name. Similar to `namespaces` on Linux, this abstraction provides some form of virtualization as the compartment engine can set the mapping of names to capabilities for each compartment individually. For instance, two compartments could both see a capability named "`/usr`", but for each compartment the capability linked to the name could grant access to a different file system service. This mechanism enables MettEagle to present different views of the system to compartments. Each compartment on L4Re has access to

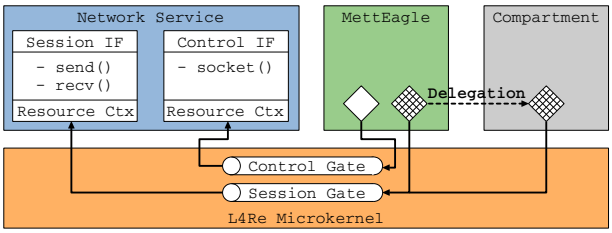


Figure 2: Example for an L4Re system service exposing different interfaces (IF) in multiple sessions (white boxes). The diamonds represent capabilities to the respective IPC gates.

a `namespace` containing its initial capabilities. `Namespaces` can be extended at runtime by adding new mappings to them, e.g. for capabilities that a compartment receives during its lifetime. Contrary to Linux' `namespaces`, which require to be integrated for various resource types, L4Re's implementation uses capabilities — a generic OS mechanism — to provide namespaces for any kind of resource.

System Call Restriction: Dedicated mechanisms for restricting the system interface available to a compartment (such as `seccomp-bpf` on Linux) are not required with L4Re. Instead, system services limit the interface they offer to a certain group of clients via dedicated IPC gates. This is possible because a system service in L4Re can securely identify a client based on the IPC gate used by the client for sending messages [37]. As shown in Figure 2, system services often expose a dedicated IPC gate for invoking control plane operations like the creation of a new session. In the context of compartments, only the compartment engine gets access to such gates, as untrusted applications running inside a compartment must not be able to spawn arbitrary sessions and thus get access to arbitrary resources, e.g. by creating unrestricted sessions. The compartments receive capabilities to session-specific gates that merely expose data-plane operations such as sending or receiving data from the network. This scheme allows for a fine-grained limitation of the API accessible to a compartment.

Resource Budgets: The secure identification of clients furthermore allows L4Re services to implement resource constraining mechanisms similar to Linux' `cgroups`. Usually, system services on L4Re create a dedicated IPC gate for each client session. During a session creation request, the caller specifies the resource restrictions of the new session and receives the capability to a session-specific IPC gate in return. Depending on the type of service, the session resource restrictions could be a bandwidth limitation, a priority, or a limit for memory allocations. The system service reflects the resource restrictions of a session by attaching a resource consumption context to the session's IPC gate as shown in Figure 2. Whenever a request arrives via a certain gate, a service first decides on whether to admit it by examining the remaining resource budget in the resource consumption context of the corresponding IPC gate.

As explained in Section 3.2 the compartment engine takes care of creating sessions with each system service required by a compartment before launching it. While doing so, the compartment engine adds a description of the respective resource limits to each session creation request and delegates the resulting capabilities to the new compartment. Reusing a session for multiple compartments and thus letting them share a common resource limit is possible as well. Note that unlike with `cgroups`, there is no uniform resource constraining mechanism across all L4Re system services. Instead, our compartment engine provides a transparent way to translate a compartment's resource limits into service-specific session creation requests.

4 Implementing a Prototype Compartment Infrastructure

This section describes the implementation aspects of components necessary for supporting compartments on L4Re.

4.1 The L4Re Compartment Environment

One option to implement the services of the compartment environment would be the use of a paravirtualized Linux (*L4Linux*) VM running on top of the L4Re base system. L4Re integrates such an L4Linux instance as a system service, allowing the latter to donate functions like device drivers to other L4Re applications [31].

In the context of compartments however, we chose to reimplement the required system services as native L4Re applications to avoid the drawback of the huge TCB that is entangled with using a single Linux VM as a provider for functionality. Mitigating this problem would be possible by spawning a dedicated L4Linux VM per compartment, but we discarded this approach as it would foil the benefits of lightweight virtualization. We thus started from L4Re's public development repository[1] and extended the L4Re base system with the multiple components to enable compartments.

First, we wrote an in-memory file system service called `SPAFS`. Unlike L4Re's ROM FS, `SPAFS` has write support and implements directories. These changes also required extending L4Re's VFS to support `SPAFS` functionality.

In order to make the MettEagle service accessible remotely, we equipped L4Re with a native network stack. To this end, we ported the driver for a 10 GBit Ethernet NIC from the Ixy driver framework [20] to L4Re. We also designed and implemented `LUNA`, a network service that multiplexes a NIC between multiple applications and implements a simple UDP/IP stack.

For the purpose of demonstrating compartments with an end-to-end benchmark, we integrated a simple FaaS agent into the Phlox high-level runtime. Phlox allows remote users to execute

[1] https://github.com/kernkonzept

code on an L4Re system. To do so, a function needs to be registered with MettEagle by uploading a code archive and assigning a name to it. Later on, users can instruct MettEagle to execute this function with input sent over the network. MettEagle sandboxes each function instance in a separate compartment.

Often, FaaS clients use interpreted languages for implementing function code [16]. We thus ported `python3` to L4Re to offer some form of standardized and widely used runtime that is also found with major cloud providers. `Python3` for L4Re was built using a novel cross-compiler that generates L4Re executables from standard Linux source packages.

Lastly, we implemented parallelized versions of both `moe`'s memory management (called `LSMM`) and its ROM FS (dubbed `PROMFS`) to increase the performance of L4Re on large machines. To this extent, we ported the LLFree page frame allocator [55] to L4Re and wrapped it in a user-level service in charge of managing the majority of the system's memory. The use of both `LSMM` and `PROMFS` is transparent to compartments, letting them act as a refinement of selected services usually offered directly by `moe`. We also enabled `moe` to run on multiple cores, avoiding cross-core IPC when interacting with it. As L4Re implements cross-core IPC synchronously using inter-processor interrupts (IPIs), this modification increases the overall system performance. However, unlike with `LSMM` and `PROMFS`, the internals of `moe` are still serialized and protected by a single lock.

4.2 Implementing Compartment Restrictions

As pointed out in Section 3, MettEagle implements visibility restrictions and virtualization by providing a minimal set of capabilities to compartments. If a compartment requires networking, MettEagle spawns a dedicated session with `LUNA` and passes the resulting capability to the compartment. `LUNA` sessions are bound to a specific address upon creation, forcing the compartment to use a fixed network endpoint.

For each compartment, MettEagle spawns a separate file system instance for data that the respective compartment needs to modify. In the future, this could be optimized by implementing overlay file systems. Read-only files (e.g., the `python` binary for executing the scripts of function) are hosted in a file system shared between all compartments.

Resource restrictions are implemented using quotas. In the current implementation, MettEagle can configure limits on resources like CPU allocation, memory consumption, or network bandwidth for each compartment. The concrete resource restriction mechanism is implemented in the system service that manages a resource. For instance, the scheduler in the L4Re microkernel allows to configure the set of CPU cores the threads of a task can use. MettEagle leverages this mechanism to restrict the CPU core allocation of a compartment according to a user-defined policy.

4.3 Lessons Learned

When implementing MettEagle, we came across a number of performance issues in the implementation of L4Re. We describe selected issues here which we consider generally insightful for the development of microkernel-based OSes.

Callbacks and Resource Pools: It turned out that L4Re's policy of having only a single reply capability limits performance. As the implicit reply capability of a thread voids when receiving the next IPC message, a server thread in L4Re can only serve one client at a time. Thus, each request arriving at the compartment service requires spawning a new thread for the execution and monitoring of a compartment. We avoided this overhead by using a callback mechanism. Upon invoking the compartment service, a client specifies a capability to an IPC gate to which the service communicates the result of the operation. The client then waits for messages on this gate to receive an answer from the compartment service.

Resource Reclamation on Critical Path: We observed that `unmap` (i.e., capability revocation) operations dramatically slow down the system when occurring on the critical path. The reason for this is the RCU-based implementation of the L4Re kernel. Threads revoking a capability can be blocked for an entire scheduling tick (10 ms by default) to ensure that a safe time point for deleting a kernel object is reached. Consequently, we implemented the compartment service and the auxiliary system services in a way to never carry out revocations on the hot path. We mitigated this issue by reusing system resources. For example, in the compartment service, instead of spawning new threads for monitoring the exit status of a compartment, we deployed thread pools to avoid the recurring overhead of spawning and deleting monitoring threads. Likewise, we carried out deletion operations in dedicated threads that do not block the actual service interfaces.

Locking of Capability Data Structures: We noticed a scalability bottleneck in `map` and `unmap` operations (i.e., capability delegation and revocation). For a delegation, the task structures of the source and destination task were locked, preventing parallel mapping from one task to multiple other tasks. We removed the lock for the source task due to the fact that the affected capabilities are already locked in the kernel's capability data structures. A similar optimization could be done for the `unmap` operation. However, even with these optimizations `map` and `unmap` do not scale well for highly parallel workloads, requiring further improvements. This demonstrates that even on a microkernel, sophisticated fine-grained locking strategies are needed to provide good performance for parallel workloads.

Component of MettEagle	SLOC
L4Re kernel	41,406
Sigma0 (root pager)	1,249
Moe (root task)	4,810
Ned (system loader)	2,870
IO (PCI driver)	22,012
LSMM (parallelized memory manager)	4,289
PROMFS (parallelized boot FS)	780
SPAFS (file system)	501
LUNA (NIC driver & UDP stack)	8,735
Compartment Service	1,793
Phlox (FaaS launcher)	826
Σ	89,271

Table 1: TCB size of MettEagle container infrastructure.

Component of Linux	SLOC
Linux kernel v6.7.4 (including ramfs)	1,454,290
NIC driver module	31,675
containerd	922,901
runc	290,946
Σ	2,699,812

Table 2: TCB size of Linux container infrastructure.

5 Security Evaluation

To the best of our knowledge, there is no hard and precise metric on the *trustworthiness* of a computing base. First, by the very nature of it, one does not know how many undiscovered security bugs are present in a given system. Second, it is hard to judge how many of these security bugs would be (partially) mitigated by the isolation architecture of a given platform. And third, the real-world security posture of a code base also depends on the activeness of development and the thoroughness with which bugs are investigated.

Because of these difficulties, we decided on using two proxy metrics as evidence that a container infrastructure based on a microkernel is more secure compared to Linux implementations. We provide a rough estimate of overall TCB *size* by counting lines of code and a vulnerability study based on past CVEs. At the end of this chapter we furthermore give insights on how a microkernel-based container architecture can mitigate timing-based attacks.

5.1 Trusted Computing Base Comparison

Table 1 and Table 2 list the TCB sizes of the MettEagle and Linux container implementations. We show source lines of code (SLOC, generated using SLOCCount [53]) as an established proxy metric for TCB complexity. Although SLOC does not directly measure the likelihood of code vulnerabilities, it gives an order-of-magnitude impression of the relative differences between the two systems. A significantly smaller number of code lines indicates that a software project might contain less bugs and it might be easier to uncover bugs as there are fewer moving parts. For our approach, we list the L4Re kernel and the userspace components `sigma0`, `moe`, `ned`, and `IO` as fundamental L4Re services. The network service `LUNA`, the `SPAFS` file system, and both MettEagle services constitute new components developed for this paper. On the Linux side, a kernel 6.7.4 was measured, limited to those C files actually contributing to the running kernel binary. In addition, the NIC driver module is listed. Contrary to L4Re, these components all run in kernel mode. We also mention the two pieces of container infrastructure `containerd` and `runC` (Go code counted using cloc [17]), although caveats apply as these components clearly offer more functionality compared to our research prototype. However, splitting that functionality in multiple isolated components, where only some are part of the core infrastructure TCB, may be possible.

In general, the microkernel implementation runs less code in CPU privileged mode compared to Linux. The small size of the kernel makes it easier to reason about it as part of a certification process. For instance, L4Re is certified according to both security (EAL4+ [13], SECRET level by the the German Federal Office for Information Security [12]) and safety (ASIL B [26]) standards. Outside the kernel, microkernel-based systems benefit from the absence of an all-powerful root account, and capability-based access control which encourages a system design following the principle of least authority.

5.2 Vulnerability Study

The small TCB size and privilege-reduced components of MettEagle should result in a better security posture. To demonstrate this point, we conducted a study of existing vulnerabilities in container infrastructure on Linux. Our approach is inspired by Biggs et al. [11]. We evaluated Common Vulnerabilities and Exposures (CVEs) collected by NIST [46], which describe known cybersecurity vulnerabilities in present and past software versions and assign vulnerability scores. The scoring follows the Common Vulnerability Scoring System (CVSS) [3], ranging from 0 (low) to 10 (critical).

To find relevant CVEs, we searched the NIST database for the terms ‘linux kernel seccomp’, ‘bpf’, ‘ebpf’, ‘linux kernel namespace’, and ‘linux kernel cgroups’. We limit our study to isolation compromises between containers, which we consider to be the highest impact vulnerabilities. The most prominent avenue are severe kernel-level exploits, so we filter our search results for Linux kernel exploits with high (CVSS 7.0–8.9) and critical (CVSS 9.0–10) scores according to CVSS scoring system v3.1.

The resulting list of 33 vulnerabilities is shown in Ta-

CVE	Cat.	Eval.	CVE	Cat.	Eval.	CVE	Cat.	Eval.
CVE-2023-39191	sc	FM	CVE-2021-4197	cg	PM	CVE-2017-18509	ns	FM
CVE-2023-35001	ns	PM	CVE-2021-3600	sc	FM	CVE-2017-18367	sc	FM
CVE-2023-31248	ns	PM	CVE-2021-3493	ns	PM	CVE-2017-17450	ns	N
CVE-2022-34918	ns	PM	CVE-2020-25220	cg	PM	CVE-2017-17448	ns	N
CVE-2022-32250	ns	PM	CVE-2019-11815	ns	N	CVE-2016-4805	ns	PM
CVE-2022-30594	sc	FM	CVE-2019-10639	ns	PM	CVE-2016-2853	ns	PM
CVE-2022-24122	ns	N	CVE-2019-9893	sc	FM	CVE-2016-1576	ns	PM
CVE-2022-1055	ns	FM	CVE-2019-2054	sc	FM	CVE-2015-1328	ns	PM
CVE-2022-0492	cg	PM	CVE-2018-16884	ns	PM	CVE-2015-8709	ns	FM
CVE-2022-0185	ns	PM	CVE-2018-18955	ns	FM	CVE-2014-1733	sc	FM
CVE-2021-4204	sc	FM	CVE-2017-1000111	ns	PM	CVE-2011-2189	ns	N

Table 3: List of CVEs classified by category (Cat.) and evaluation (Eval.). The categories consist of `seccomp` and / or `eBPF` (sc), `namespaces` (ns) and `cgroups` (cg) on Linux. The evaluation is marked as FM (fully mitigated), PM (partially mitigated), and N (not mitigated).

ble 3. This number may seem small, but these are potential high-severity exploits as they directly affect the Linux kernel. Mistakes in the userspace container infrastructure may contribute additional vulnerabilities, but these are not considered here as our FaaS runtime is not feature-comparable to the Linux implementation. For the 33 vulnerabilities considered, we categorize our findings the following way:

Fully Mitigated by MettEagle (FM): We do not see a possibility how a similar exploit could be constructed on our microkernel-based system.

Partially Mitigated by MettEagle (PM): If a similar implementation mistake is made, a similar vulnerability may exist in our system. However, immediate, total container-isolation compromise would not occur.

Not Mitigated by MettEagle (N): Our system would be affected the same way as the Linux system. In the following, we will discuss some sample vulnerabilities individually.

5.2.1 seccomp-bpf

`Seccomp-bpf` uses the extended Berkeley Packet Filter (eBPF), to restrict available system calls at kernel level [1]. With eBPF being a code interpreter at kernel level and `seccomp-bpf` using eBPF to express its filter rules, security issues can arise from flaws in either `seccomp-bpf` itself or the underlying eBPF. MettEagle does not require such filtering mechanisms. Only interfaces, for which capabilities are granted are visible to a container. To bypass this access control, the capability implementation itself would need to be compromised, which is part of the microkernel. While implementation bugs in the capability management may exist, we consider these less likely due to the lower complexity compared to a complete code interpreter and checking mechanism as required for `seccomp-bpf`. To substantiate this claim we gathered the SLOC count for the BPF directory in the Linux source. We counted 50,200 lines of code which is more than our entire microkernel (41,406 SLOC). While the SLOC of the BPF directory includes BPF debugging tools in addition to the eBPF interpreter and checker, we note that the microkernel also includes much more than the capability management. We found six CVEs directly related to `seccomp-bpf` and two CVEs related to eBPF in the Linux kernel. We argue that all of them are fully mitigated by MettEagle as the capability system provides `seccomp-bpf` functionality.

5.2.2 namespaces

We found 22 CVEs related to Linux `namespaces`. Out of these, we classified four as fully mitigated by MettEagle, 13 as partially mitigated, and five as not mitigated. Two more CVEs are actually related to `cgroups` and are therefore included in Section 5.2.3. As an example, we consider CVE-2018-18955 to be fully mitigated by MettEagle. It describes a vulnerability in the implementation of `namespace` restrictions, allowing privilege escalation via mishandling of nested user `namespaces`. In L4Re, resource access is enforced by capabilities, so such an escalation can only occur by compromising the capability system itself. Other vulnerabilities are only partially mitigated. An example is CVE-2018-16884, a vulnerability in the Linux kernel's NFS41+ subsystem, in which NFS41+ shares are simultaneously mounted in different network `namespaces`. We consider this to be a bug in the NFS component, causing it to misconfigure the isolation afforded by `namespaces`. Such misconfigurations by a resource provider component like NFS can occur in MettEagle as well. However, this specific CVE on Linux causes memory corruption and privilege escalation in the kernel. On MettEagle, such a mistake would only compromise a userspace component and thus does not immediately allow containers to access each other's memory. Inter-container isolation on MettEagle still holds, thus our ranking as partially mitigated.

CVE-2022-24122 is an example of the third category, *not mitigated*. It describes a use-after-free bug in the implemen-

tation of unprivileged user namespaces. This could lead to compromise of the Linux kernel. Equivalently, the L4Re could contain a memory bug in the code for capability management, compromising the kernel. However, the smaller size and certification of L4Re suggest that these kinds of bugs are less likely to occur.

5.2.3 cgroups

We found three CVEs related to Linux `cgroups`. These can all be partially mitigated by MettEagle. CVE-2022-0492 for instance describes an exploit, where a container can escalate privileges and bypass `namespace` isolation due to a flaw in a `cgroups` feature, allowing to execute arbitrary code with root privileges. In MettEagle, resource restrictions are implemented by resource contexts in userspace components. As these components may exhibit similar implementation flaws, resource restrictions may be equally evadable. However, such a compromise would only affect a single resource and would not result in breaking the kernel or inter-container memory isolation due to root privilege escalation as there is no all-powerful root on a capability-based OS.

5.3 Timing-Based Attacks

In this section, we discuss classes of security problems that are related to time, which are underrepresented in the CVE analysis. These issues include performance-related denial-of-service and timing side-channel attacks. In general, these problems can occur when an attacker and a victim share some state in a system for which timing properties can change based on the behavior of tenants.

One example of such an attack is KernelSnitch [41]. This attack can leak process information by observing the variance in access time to Linux kernel data structures. This variance in access time is due to the variable occupancy of such structures depending on the behavior of other processes. Similarly, Patel et al. [47] use this correlation to attack the performance of victim processes. They strategically trigger worst-case execution time of operations on kernel data structures. The attack described by Mergendahl et al. [44] targets a different, but also shared, component, the scheduler. The authors discovered that a low-priority adversary can arbitrarily delay a specific scheduling instance of an attacked thread by exploiting non-constant operation latency in scheduling data structures on seL4.

There are also shared data structures in L4Re, like the scheduler or capability inheritance trees. Consequently, similar attacks might also exist for L4Re. However, two properties of L4Re make it more resilient to these kinds of attacks. First, the L4Re kernel is real-time capable. Thus, many kernel operations are guaranteed to have bounded execution time, which reduces the likelihood of cross-process timing influences. Second, the microkernel approach offers fewer opportunities for such attacks than traditional systems as the amount of data structures that applications inevitably share can be reduced by careful system design, e.g., by using a separate instance of each system service per compartment. A downside of such mitigation strategies is their increased memory demand. However, the extent of this overhead is to be evaluated in future studies.

The idea of using separate system service instances for each container can also be extended to the kernel itself. For example, Ge et al. [25] implement temporal isolation of applications by running them in fully cloned kernel instances. Thus, we believe that the architecture of MettEagle can provide a better resilience against timing-based attacks compared to Linux-based approaches.

6 Performance Evaluation

In the following, we compare the implementation of compartments on L4Re with standard solutions for lightweight virtualization from a performance perspective. As we want to focus on the bare OS mechanisms, i.e. the isolation and virtualization of containers, we did not deploy warm start optimizations such as provisioning of standby containers with pre-initialized runtimes. In our evaluation we answer the following questions:

- What is the startup time of compartments on L4Re compared to their Linux counterparts?
- Is it possible to build I/O services on L4Re with a performance competitive to monolithic systems?
- What is the end-to-end performance of an application benchmark using standard runtimes running in L4Re compartments?

As a comparison for L4Re compartments, we measured Linux stock processes, runC [32], and Kata Containers [23] with Firecracker [7]. While Linux processes do not guarantee container-grade security properties, they represent an optimal baseline with respect to performance. RunC is the OCI's reference implementation for a low-level container runtime on Linux. It deploys the aforementioned Linux mechanisms like `seccomp-bpf`, `cgroups`, and `namespaces` to create sandboxed execution environments. Firecracker is a lightweight VM framework that builds on top of KVM. It is used with cloud providers like AWS. In our setup, both runC and Firecracker use containerd [24] as high-level container runtime. To integrate Firecracker in a containerd-based system we used Kata Containers [23] as a backend.

6.1 System Setup

All measurements were done on two identical dual-socket servers that are equipped with two Intel Xeon Platinum 8358 CPUs (32 cores per CPU) and 500 GiB of main memory each. For all benchmarks, we disabled both hyperthreading (SMT) and temporary overclocking (TurboBoost). We furthermore

set the CPU's pstate configuration to maximum performance mode. Additionally, the servers both feature a 10 Gbit (Intel 82599 / Intel X540) Ethernet NIC.

The results for benchmark setups running on Linux were recorded using kernel version 6.7.4, runC version 1.1.10, containerd version 1.7.9 and Kata version 3.3.0 with a small patch to enable the measurement of startup times in Firecracker. As `SPAFS` currently lacks a disk backend, we used in-memory file systems on all platforms.

6.2 Container Startup Latency

First, we measured the startup latency of a container. In this benchmark, the containers started consist of a single process and do nothing but recording a timestamp directly after booting. For runC the startup latency reported includes the setup of `cgroups`, `seccomp` filters, and file systems. To measure best-case performance, we set empty `seccomp` filters. For Kata Containers with Firecracker we measure the time from starting the microVM until the container inside prints its first message. To ensure a fair comparison, meta operations for spawning a compartment on L4Re, such as creating a new session with the MettEagle service, contribute to the startup latency.

Figure 3 shows the cumulative distribution function (CDF) of container cold start latency when starting multiple instances in parallel. The benchmark launches N containers at once and then waits for all containers to terminate before starting the next iteration. Note that in practice, many workloads are less extreme with regards to concurrency and rather launch containers in a staggered fashion, making the results shown in Figure 3 an estimation of worst-case performance.

Linux processes have an almost constant startup time, independent of the number of processes spawned in parallel (roughly 200 μs). Similarly, Kata Containers remain constant in their startup latency, albeit at a higher level, as the setup of a VM is more costly than launching a process. On L4Re, it takes 1 ms to start a container on an empty and idle system (N = 1). This startup latency gradually increases when more containers are launched in parallel peaking at 100 ms for 64 parallel launches. With runC, it takes around 70 ms to start a single container. Like with L4Re compartments, this number grows with increasing parallelism, reaching up to 200 ms when starting 64 containers in parallel.

The difference between the cold start latency of a Linux process and an L4Re compartment for N = 1 largely stems from the fact that L4Re loads two binaries when starting a task. Due to the implementation of L4Re's memory management, a task always starts with executing a common helper binary. This helper runs in the first thread of a task and in turn loads the binary of the actual application, effectively doubling the startup overhead. Note however, that this procedure is not intrinsic to a microkernel OS and could be optimized e.g. by running the helper's code directly in the application threads.

The reasons for the growing startup latency of L4Re compartments when facing increased parallelism are rooted in system components not optimized for parallel workloads. We verified that the increase in startup latency is not a consequence of growing IPC latency. However, the L4Re kernel uses coarse-grained locking on its internal data structures which affects the performance of carrying out capability delegation and revocation operations in parallel, which occur a lot when spawning a compartment. A similar issue can be found with `moe`, which uses a single lock to protect its internals. While `LSMM` and `PROMFS` parallelize the memory management and the boot file system, `moe` is still needed for creating objects like scheduling quotas, causing another bottleneck.

In addition to our previous benchmark, Figure 4 shows the startup latency of a single container as a function of idle containers that are present in the system. In this context *idle* means that all threads of the respective containers are blocked. The container whose startup time is measured and the idle containers are not started in parallel. Figure 4 thus gives insight into whether the management data structures of a specific container engine or OS themselves become a bottleneck when many containers are present. The results of the benchmark show that on all platforms, the number of idle containers does not significantly impact their startup latency.

6.3 I/O Performance

To evaluate I/O performance, we compare the network performance available to L4Re compartments to that of Linux processes and containers. We measured the latency of a UDP-based ping-pong benchmark over a 10 GiB/s Ethernet interface. The remote host that was pinged ran Linux, while the latency was recorded on the respective platform. On all platforms the latency was roughly the same with an average value of 40 μs.

Figure 5 shows the bandwidth achieved using multiple sockets in parallel. As expected, Linux and runC perform similarly as they use the same network stack. With a low degree of parallelism, Linux achieves a higher bandwidth than L4Re (900 MiB/s vs. 350 MiB/s) since L4Re does not implement advanced NIC features such as receive side scaling, and all driver processing is done on a single core, which is also used by the first thread of the benchmark. With increasing parallelism and thus more threads sending traffic from different cores, this effect diminishes. While L4Re was able to achieve throughput at line rate using many sockets, Linux performance decreased when using the NIC in parallel. Our efforts to tune Linux' network stack did not significantly improve its performance. This is surprising as a bandwidth test using `iperf` [21] achieved line rate throughput for many parallel TCP connections but yielded results similar to our custom benchmark when using UDP.

6.4 Application Benchmark

We chose SeBS [16], a client for benchmarking serverless execution environments, to simulate an application workload.

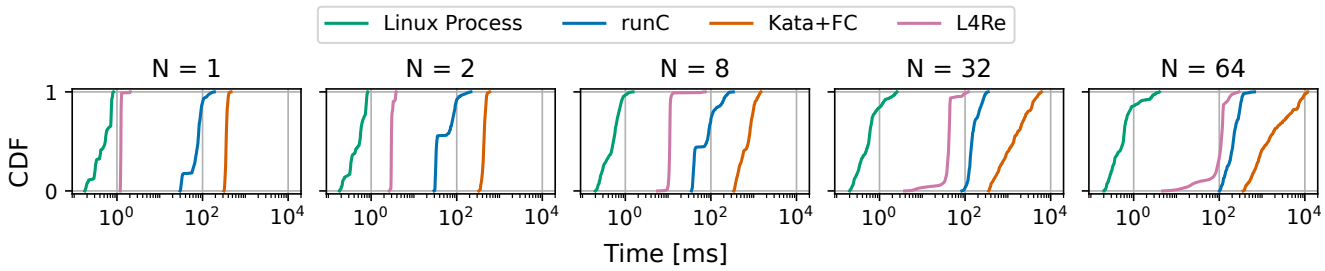


Figure 3: Cumulative distribution function of the container startup latency when launching N containers in parallel. The leftmost subplot (N = 1) shows the startup latency of a single container on an empty system.

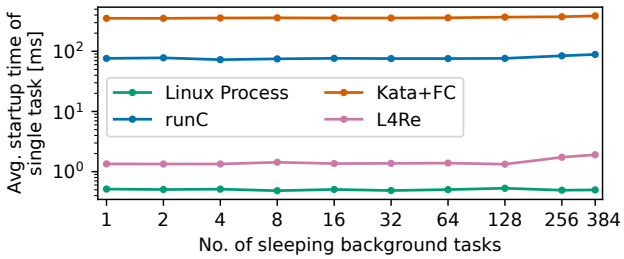


Figure 4: Average startup latency of a single container as function of the number of idle containers present in the system.

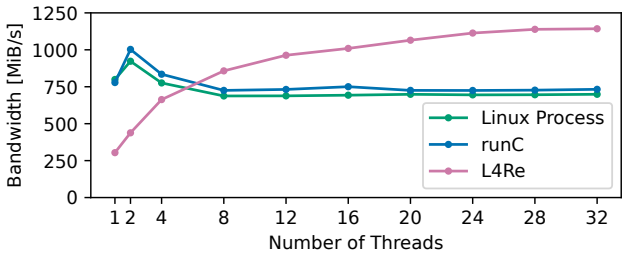


Figure 5: Bandwidth for transmitting data over a 10 GBit Ethernet interface (using 1472 B UDP packets) as a function of the number of parallel data streams.

SeBS deploys a set of benchmarks written in `python` to benchmark a FaaS server running on a remote machine. To avoid performance advantages for L4Re due to missing functionality, we implemented service similar to MettEagle's FaaS agent on Linux. This custom FaaS agent executes the benchmarks as a process, a container, or a Kata Container using a Firecracker backend respectively.

The use of a `python`-based benchmark yields multiple insights: First, it demonstrates that L4Re is capable of running a standardized runtime. This allows for executing unmodified code written for a platform like Linux on L4Re, giving a hint on how to deploy existing applications in compartments. Second, `python` is a program that makes heavy use of OS services. During startup, `python` interacts with dozens of files and performs a lot of memory allocation and deallocation. Due to their intrinsic communication overhead, microkernel OSes are known to have performance deficits for such operations. Hence, from a microkernel perspective, the use of a `python` benchmark approximates a worst-case example with respect to performance, allowing us to explore L4Re's limits.

In the following, we present results for a subset of the SeBS benchmark suite. We omitted benchmarks for uploading data to cloud storage, as this workload does not relate to our work. Also, we refrained from porting both the `pytorch` machine learning library and `squiggle`, a python library for DNA visualization to L4Re. Both benchmarks are compute-heavy and thus do not exercise the operating system. Lastly, we also had to omit an FFMPEG benchmark since it assumed support for the `fork` system call which does not exist on L4Re. In the end, we evaluated benchmarks for executing an empty `python` function, HTML manipulation, file compression, and several graph processing algorithms, whose dependencies were ported to L4Re.

Whenever possible, we ran the SeBS benchmarks with both a standard problem size as well as with a *tiny* problem size. While the tiny size is too small to represent a real-world use case, it provides us with an approximation for the runtime initialization overhead introduced by `python`, which differs for each benchmark due to a different set of `python` modules being loaded. The empty function and the file compression benchmark do not offer a configurable problem size.

Figure 6 depicts the latency for sequentially executing the SeBS benchmark. For all benchmarks except ZIP, the end-to-end performance of L4Re compartments is at most 15% worse than that of runC. When running the HTML benchmark, L4Re was 10% faster than runC. The right subplot reveals that the pure execution time spent in `python` (execution time of python code with module loading, excluding the startup latency of the core interpreter) is higher with L4Re compartments than on any other platform. However, the fast startup latency of compartments levels out this disadvantage, leading to a comparable end-to-end latency of function execution. For all benchmarks, the end-to-end latency for Kata was the highest which we attribute to the slower startup of

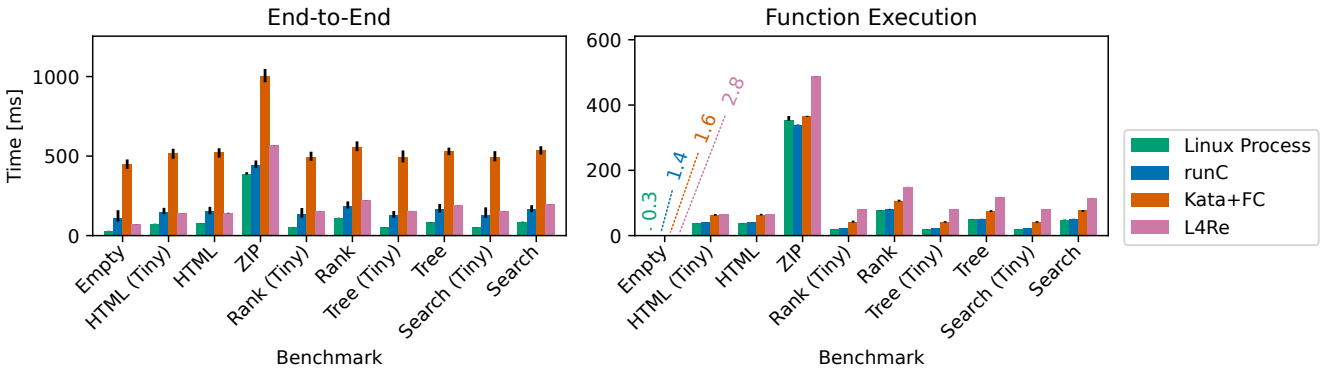


Figure 6: Median latency for sequential execution of the SeBS serverless benchmark. The left subfigure depicts end-to-end latency as observed by the client, while the right subfigure shows the execution time in python. Benchmarks run with the tiny size are marked by the *Tiny* suffix. The whiskers show the 5th and 95th percentile, respectively.

the VM and the fact that the container images Kata is copying into the VM upon initialization are much larger for a `python` benchmark than those used in the microbenchmark.

Interestingly, the absolute difference in execution time between L4Re and the other platforms is the same for both benchmark sizes, regardless of the benchmark type. For instance, the latency difference between running the page rank benchmark (Rank) in L4Re compartments and runC is around 45 ms for both the tiny benchmark size (graph with 10 nodes) and the standard size (graph with 10,000 nodes). Since the benchmark function script itself loads several modules and this time is included in the function execution latency, it is likely that the performance disadvantage of L4Re is caused by a slower initialization of `python` modules. This in turn indicates that the execution of `python` code on L4Re is not slow per se.

While investigating the numbers for `python` on L4Re, it turned out that L4Re's file system performance is much worse compared to Linux. For instance, a single call to `stat` takes around 4 μs on L4Re, but only 460 ns on Linux. This roughly tenfold performance disadvantage also applies to other file-related operations such as `open` or `read`. As even running an empty `python` script involves hundreds of calls to these functions, L4Re's shortcomings in file system performance strongly influence the benchmark as a whole. This also explains the weakness of L4Re in the ZIP benchmark, which executes many file operations also after the initialization phase.

Lastly, Figure 7 shows the results for running the SeBS benchmark in burst mode. In this mode, the client triggers 16 function invocations in parallel, leading to the parallel execution of 16 container instances with `python` on the server side. In general, all platforms performed worse than in the sequential setting. Comparing runC and L4Re, the end-to-end latency for an empty function and for the HTML benchmark is again similar. The ZIP and the graph processing benchmarks, however, take one to two times longer on L4Re than on runC. Yet again, the performance deficits are not caused by the compartment mechanism itself, but rather by the execution of the `python` code. However, L4Re consistently outperforms Kata containers.

As with the sequential pattern of the SeBS benchmarks, our file system implementation accounts for a large share of the performance deficits of L4Re compared to runC. The increase in function execution latency of the graph benchmarks (rank, tree, and search) is caused by the dynamic loading of shared objects required by the `python` scripts. Surprisingly, loading shared objects from within different tasks in parallel appears to slow this procedure as a whole. The exact reason for this is still subject to investigation.

7 Discussion

As the performance evaluation in Section 6 has shown, the implementation of containers on microkernels is feasible. Also on large machines, a microkernel-based container framework is able to provide performance competitive to standard solutions for most workloads. While the microkernel architecture introduces some intrinsic overhead, we believe that implementation optimizations could further improve the performance of L4Re compartments significantly. For example, L4Re's file system overhead could be reduced by deploying a file system design similar to approaches used with persistent memory where most file operations are implemented as accesses to shared memory instead of IPC [15].

Also, the FaaS agent of MettEagle currently lacks warm start optimizations [8, 19] found in current production-grade lightweight virtualization solutions. We do not see conceptual limitations that would prevent the adaptation of these optimizations with the compartment framework of L4Re. In fact, microkernels conceptually offer means of optimization even beyond those of monolithic kernels. On a microkernel-based system, different compartments could be using separate

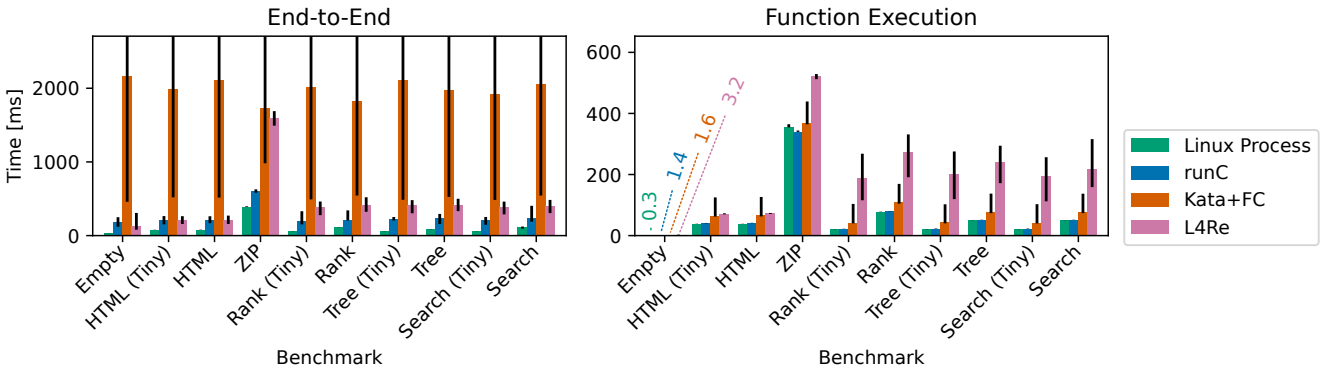


Figure 7: Median latency for starting 16 functions of each SeBS benchmark in parallel. The left subfigure shows end-to-end latency as recorded at the client. The right subfigure depicts the pure execution time in python. Benchmarks run with the tiny size are marked with a corresponding suffix. The whiskers denote the 5th and 95th percentile. The whiskers for Kata+FC in the left subplot extend to values between 3000 and 4000 ms but have been cropped to improve readability of the plot.

instances of system services such as storage or network stacks. Each instance could be tuned to its specific container workload, offering bespoke interfaces and abstractions. We did not explore such optimizations in MettEagle, but its overall architecture can be adapted to support such container-specific service instances.

In terms of interfaces, MettEagle currently does not support the execution of compartments loaded from OCI-compliant images. With an effort to enhance L4Re's POSIX Compatibility Layer, there are no fundamental limitations that prevent supporting OCI images on L4Re. However, this requires a lot of implementation and standardization effort, as OCI containers rely on many Linux-specific features like certain *sysfs* files.

Lastly, we believe that the findings of this paper could also be transferred to other microkernel platforms. For instance implementing containers on seL4 would work similarly on a conceptual level and grant additional security benefits as the seL4 kernel is formally verified [35]. The researchers developing seL4 also showed that their system is able to provide protection against a variety of timing-based side-channel attacks [28], which is of particular concern when running containers of mutually distrusting clients on the same hardware.

8 Related Work

The emergence of microservices motivated research for optimizing lightweight virtualization with respect to performance and security. For container-based systems, Cntr [51] introduced a mechanism for dynamically adding rarely used tools to production container images, reducing their size and deployment time. On the security side, dedicated interest groups work on hardening containers on Linux [40]. Additionally, chestnut [14] allows for automatically generating `seccomp-bpf` filters from application binaries, only granting access to system calls the application needs. Many such Linux-related container-hardening techniques become superfluous with MettEagle because a capability-based approach like L4Re promotes adherence to the principle of least authority.

Nakata et al. proposed to mitigate attacks exploiting the weaker security guarantees of containers by transplanting Linux containers across different OS types, thus thwarting attacks that only work when running on specific kernels [45]. The same reasoning applies to running containers on L4Re.

BlackBox [29] implements secure container execution on an untrusted operating system. Containers are shielded against the operating system using virtualization and system call sanitization. Although BlackBox achieves a small TCB for confidentiality, the otherwise large, untrusted operating system needs to be relied on to guarantee availability. The same applies to MettEagle, albeit with a much smaller TCB contributing to availability.

In pursuit of increasing the security of lightweight virtualization, lightweight VMs gained traction. Shen et al. used a library OS as a guest operating system running in lightweight VMs, increasing their performance by saving mode and context switches in the guest [49]. Their approach also reduced the interface between the guest OS and the hypervisor, shrinking the attack surface of the latter. For frameworks like Kata, Li et al. proposed runD [39], a set of optimizations for reducing the performance impact of container isolation mechanisms like `cgroups` when an application is already isolated in a lightweight VM. Manco et al. demonstrated the use of unikernels together with carefully designed VM infrastructure, yielding startup times and application performance even better than that of containers on Linux [42]. We believe that MettEagle also offers a high degree of isolation, like VMs do. The attack surface between containers and the microkernel is far smaller than under a monolith, like Linux. But in contrast to unikernel-based VMs, containers in MettEagle have native access to

operating system features like multithreading for performance and address-space-based compartmentalization for security.

Lastly, Koller and Williams argue that FaaS applications are less dependent on the Linux environment than traditional applications [36]. As an example, Szekely et al. required little effort to port FaaS applications to their new σOS API [50]. We argue in a similar way that porting real-world FaaS code to MettEagle will be manageable and not hinder adoption.

9 Conclusion

MettEagle has demonstrated the feasibility of implementing containers on the L4Re microkernel. Due to the fundamental security benefits of microkernels, our container design awards a more robust security posture compared to containers on Linux. While Linux needs to implement additional restriction mechanisms for providing container-grade isolation to processes, such mechanisms are either not needed or conceptually much simpler on a capability-based microkernel, since it fully isolates processes by default. We have demonstrated this security advantage with a study of past CVEs, many of which are mitigated on L4Re because the attacked restriction mechanisms are not present.

Further, we have shown that the simplicity of microkernel-based systems can even result in improved performance. Although some performance deficiencies remain, these are mostly caused by implementation issues rather than being inherent to the microkernel-based design. While we applied only limited optimizations for server-grade workloads to L4Re, we still achieve raw container startup times outperforming Linux containers. With these results, we believe a microkernel-based cloud infrastructure is worth developing further, as the security benefits are apparent and performance is competitive.

Acknowledgments

We would like to thank our shepherd, Yuvraj Patel, and the anonymous reviewers of SOSP'24 and OSDI'25 for their helpful suggestions. This research was funded by the German Research Council DFG through grants 502457159 (FOSSIL) and 501887536 (ParPerOS). The project was furthermore partially supported by the German federal ministry of research, technology, and space (BMFTR) within the MANNHEIM CeCaS project.

References

[1] Seccomp BPF (SECure COMPuting with filters) — the linux kernel documentation. https://www.kernel.org/doc/html/v4.18/userspace-api/seccomp_filter.html, August 2018. [Online; last accessed on November 28, 2024].

[2] New seccomp mode aims to improve performance. https://lore.kernel.org/linux-security-module/c22a6c3cefc2412cad00ae14c1371711@huawei.com/t/, 2020. [Online; last accessed on November 28, 2024].

[3] CVSS v3.1 specification document. https://www.first.org/cvss/v3.1/specification-document, August 2021. [Online; last accessed on November 28, 2024].

[4] NVD — CVE-2022-0492. https://nvd.nist.gov/vuln/detail/CVE-2022-0492, 2022. [Online; last accessed on November 28, 2024].

[5] NVD — CVE-2022-30594. https://nvd.nist.gov/vuln/detail/CVE-2022-30594, 2022. [Online; last accessed on November 28, 2024].

[6] Cloud Hypervisor: A virtual machine monitor for modern cloud workloads. https://github.com/cloud-hypervisor/cloud-hypervisor, 2024. [Online; last accessed on November 28, 2024].

[7] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434. USENIX Association, 2020. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

[8] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in FaaS. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 398–415. ACM, 2023. doi: 10.1145/3552326.3567503. URL https://doi.org/10.1145/3552326.3567503.

[9] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: a study of firecracker and gVisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 101–113. Association for Computing Machinery, 2020. ISBN 9781450375542. doi: 10.1145/3381052.3381315. URL https://doi.org/10.1145/3381052.3381315.

[10] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*,

pages 45–58. USENIX Association, 1999. URL https://dl.acm.org/citation.cfm?id=296810.

[11] Simon Biggs, Damon Lee, and Gernot Heiser. The jury is in: Monolithic OS design is flawed: Microkernel-based designs improve security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360067. doi: 10.1145/3265723.3265733. URL https://doi.org/10.1145/3265723.3265733.

[12] BSI. BSI-Schrift 7164: Liste der zugelassenen IT-Sicherheitsprodukte und -systeme. https://www.bsi.bund.de/SharedDocs/Zulassung/DE/Produkte/L4Re_Secure_Separation_Kernel_VS_BSI-VSA-10624.html, . [Online; last accessed on November 28, 2024; in German].

[13] BSI. BSI-DSZ-CC-1177-2025. https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/Betriebssysteme/1177.html, . [Online; last accessed on April 16, 2025].

[14] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *CCSW@CCS '21: Proceedings of the 2021 on Cloud Computing Security Workshop, Virtual Event, Republic of Korea, 15 November 2021*, pages 139–151. ACM, 2021. doi: 10.1145/3474123.3486762. URL https://doi.org/10.1145/3474123.3486762.

[15] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with kernel-userspace collaboration. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 81–95. USENIX Association, 2021. URL https://www.usenix.org/conference/fast21/presentation/chen-youmin.

[16] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385343. doi: 10.1145/3464298.3476133. URL https://doi.org/10.1145/3464298.3476133.

[17] Al Danial. cloc. https://github.com/AlDanial/cloc, 2024. [Online; last accessed on November 29, 2024].

[18] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966. doi: 10.1145/365230.365252. URL https://doi.org/10.1145/365230.365252.

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481. ACM, 2020. doi: 10.1145/3373376.3378512. URL https://doi.org/10.1145/3373376.3378512.

[20] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User space network drivers. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019*, pages 1–12. IEEE, 2019. doi: 10.1109/ANCS.2019.8901894. URL https://doi.org/10.1109/ANCS.2019.8901894.

[21] Jon Dugan et al. iPerf. https://iperf.fr/, 2024. [Online; last accessed on December 9, 2024].

[22] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 171–172. IEEE Computer Society, 2015. doi: 10.1109/ISPASS.2015.7095802. URL https://doi.org/10.1109/ISPASS.2015.7095802.

[23] OpenInfra Foundation. Kata containers. https://katacontainers.io/, 2024. [Online; last accessed on November 28, 2024].

[24] The Linux Foundation. Containerd: An open and reliable container runtime. https://github.com/containerd/containerd, 2024. [Online; last accessed on November 29, 2024].

[25] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing OS abstraction. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 1:1–1:17. ACM, 2019. doi: 10.1145/3302424.3303976. URL https://doi.org/10.1145/3302424.3303976.

[26] Elektrobit Automotive GmbH. EB corbos Hypervisor Certified for ASIL B - ISO 26262. https://www.elektrobit.com/newsroom/el

ektrobit-continues-acceleration-toward-safe-and-secure-software-defined-mobility/. [Online; last accessed on April 16, 2025].

[27] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μkernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 66–77. ACM, 1997. doi: 10.1145/268998.266660. URL https://doi.org/10.1145/268998.266660.

[28] Gernot Heiser, Gerwin Klein, and Toby Murray. Can we prove time protection? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 23–29, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367271. doi: 10.1145/3317550.3321431. URL https://doi.org/10.1145/3317550.3321431.

[29] Alexander Van't Hof and Jason Nieh. BlackBox: A container security monitor for protecting containers on untrusted operating systems. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 683–700. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/vant-hof.

[30] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, page 22–es, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 9781450378079. doi: 10.1145/1133572.1133615. URL https://doi.org/10.1145/1133572.1133615.

[31] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An i/o architecture for microkernel-based operating systems. September 2003. URL https://os.inf.tu-dresden.de/papers_ps/tr-ioarch-2003.pdf.

[32] Open Container Initiative. RunC. https://github.com/opencontainers/runc, 2024. [Online; last accessed on November 28, 2024].

[33] Peter Van Der Kamp and Robert N. M. Watson. Jails: confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000. URL https://people.freebsd.org/~bapt/pdfdocs/papers/jail.pdf.

[34] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 139–144. USENIX Association, 2013. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim.

[35] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

[36] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 169–173, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350686. doi: 10.1145/3102980.3103008. URL https://doi.org/10.1145/3102980.3103008.

[37] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, page 25–30, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584645. doi: 10.1145/1519130.1519135. URL https://doi.org/10.1145/1519130.1519135.

[38] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Cheng-Zhong Xu. Serverless computing: State-of-the-art, challenges and opportunities. *IEEE Trans. Serv. Comput.*, 16(2):1522–1539, 2023. doi: 10.1109/TSC.2022.3166553. URL https://doi.org/10.1109/TSC.2022.3166553.

[39] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 53–68. USENIX Association, 2022. URL https://www.usenix.org/conference/atc22/presentation/li-zijun-rund.

[40] Linux Container Hardening Project. Linux container hardening. https://containerhardening.org/. [Online; last accessed on November 28, 2024].

[41] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side channel-

attacks on kernel data structures. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL https://www.ndss-symposium.org/ndss-paper/kernelsnitch-side-channel-attacks-on-kernel-data-structures/.

[42] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 218–233. ACM, 2017. doi: 10.1145/3132747.3132763. URL https://doi.org/10.1145/3132747.3132763.

[43] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[44] Samuel Mergendahl, Samuel Jero, Bryan C. Ward, Juliana Furgala, Gabriel Parmer, and Richard Skowyra. The thundering herd: Amplifying kernel interference to attack response times. In *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022*, pages 95–107. IEEE, 2022. doi: 10.1109/RTAS54340.2022.00016. URL https://doi.org/10.1109/RTAS54340.2022.00016.

[45] Yuki Nakata, Shintaro Suzuki, and Katsuya Matsubara. Reducing attack surface with container transplantation for lightweight sandboxing. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2023, Seoul, Republic of Korea, August 24-25, 2023*, pages 58–64. ACM, 2023. doi: 10.1145/3609510.3609820. URL https://doi.org/10.1145/3609510.3609820.

[46] NIST. National vulnerability database. https://nvd.nist.gov/vuln. [Online; last accessed on November 28, 2024].

[47] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, and Michael M. Swift. Using Trāt*r* to tame adversarial synchronization. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 3897–3916. USENIX Association, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/patel.

[48] Prateek Sharma, Lucas Chaufournier, Prashant J. Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 1. ACM, 2016. URL http://dl.acm.org/citation.cfm?id=2988337.

[49] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert van Renesse, and Hakim Weatherspoon. X-Containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 121–135. ACM, 2019. doi: 10.1145/3297858.3304016. URL https://doi.org/10.1145/3297858.3304016.

[50] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with SigmaOS. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695947. URL https://doi.org/10.1145/3694715.3695947.

[51] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 199–212. USENIX Association, 2018. URL https://www.usenix.org/conference/atc18/presentation/thalheim.

[52] Andrew Tucker and David Comay. Solaris zones: Operating system support for server consolidation. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium, May 6-7, 2004, San Jose, CA, USA*. USENIX, 2004. URL http://www.usenix.org/publications/library/proceedings/vm04/wips/tucker.pdf.

[53] David A. Wheeler. SLOCCount. https://dwheeler.com/sloccount/, 2024. [Online; last accessed on November 29, 2024].

[54] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association. URL https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams.

[55] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. LLFree: Scalable and optionally-persistent page-frame allocation. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 897–914. USENIX Association,

2023. URL https://www.usenix.org/conference/atc23/presentation/wrenger.

A Artifact Appendix

Abstract

The artifact contains every custom component that is needed to recreate the measurements presented in the paper. It is a bundle of source files of the individual projects created for this paper. This includes the MettEagle software and benchmark code. However, external, unmodified software (like Linux or Firecracker) is not included.

Scope

The artifact strives to achieve *artifact available*. It only contains source files and some documentation but no all-in-one script for fitting all the individual projects together. Thus, this artifact does not claim easy *functional* evaluation. Unfortunately, due to the large number of moving parts and long compilation times, setting up the benchmark environment takes some time. If any question arise while using the provided artifact, please contact the authors.

The artifact contains the code of the MettEagle FaaS runtime including the surrounding L4Re. It is thus possible to run MettEagle using this artifact. Furthermore, all the benchmark scripts for exercising MettEagle are included. Most importantly, a modified version of the SeBS benchmark suite is included to run functions on MettEagle. All code used to create comparative benchmarks on Linux is also included.

Contents

For a precise description of the provided artifacts, please refer to the `README.md` inside of the artifact.

Hosting

The artifact is available via Zenodo: https://doi.org/10.5281/zenodo.15211297.

Requirements

The artifact is designed for the x86-64 architecture. It is tested to run inside of Qemu and on the machine described in Section 6.1. The network benchmarks can either run via emulated NICs in Qemu or via connecting two physical machines.